Polynomial-time constraint satisfaction problems

Clément Carbonnel

CNRS, LIRMM

06/11/2023

Introduction

- CSP is very expressive (problems are often easy to model)
- CSP solvers are remarkably efficient
- ... but the best tool to solve a constraint network is not always a CSP solver
  - E.g. systems of linear equations, perfect matchings...
- Today: **a quick tour of the main classes of CSPs that can be solved in polynomial time**

Prerequisites:

- Christian's course: CSP, consistency
- Elementary complexity theory: P, NP, reductions

Introduction

Some constraint networks can be solved in polynomial time because...

The language is easy

Linear equations

2-SAT

...

(Part 1)

The structure is easy

Tree-like structure

Covering constraints

...

(Part 2)

A combination of the two

Perfect matching

4-colouring planar graphs

...

(not in this course)

Disclaimer

Three (controversial) assumptions:

- We care only about the **decision problem**: does this constraint network have a solution?
  - The search problem is sometimes more difficult
- Every variable starts with the same domain D. Individual variable domains are considered as **unary constraints**.
- Constraints are given in input as **tables of tuples**:

$$D = \{1, 2, 3\},\ c(x, y) \equiv (x \neq y) : \quad \begin{array}{cc} x & y \\ \end{array} \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 1 \\ 2 & 3 \\ 3 & 1 \\ 3 & 2 \end{bmatrix}$$

Part 1: language-based tractable classes

Fixed-language CSPs

A **constraint language** Γ is a finite set of Boolean functions over a finite domain D

Fixed-language CSPs

A **constraint language** Γ is a finite set of Boolean functions over a finite domain D

CSP(Γ) = CSP restricted to networks in which every constraint uses a function from Γ

Fixed-language CSPs

A **constraint language** Γ is a finite set of Boolean functions over a finite domain D

CSP(Γ) $=$ CSP restricted to networks in which every constraint uses a function from Γ

- $3\text{-SAT} \equiv$ CSP($\{$ternary Boolean clauses$\}$) over $D = \{0, 1\}$

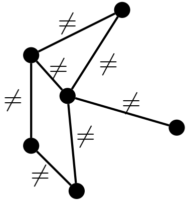
$$\ldots \wedge (x_1 \vee \overline{x_4} \vee x_3) \wedge (x_4 \vee \overline{x_2} \vee \overline{x_5}) \wedge (x_2 \vee x_1 \vee x_5) \wedge \ldots$$

Fixed-language CSPs

A **constraint language** Γ is a finite set of Boolean functions over a finite domain D

CSP(Γ) $=$ CSP restricted to networks in which every constraint uses a function from Γ

- $3\text{-SAT} \equiv$ CSP($\{$ternary Boolean clauses$\}$) over $D = \{0, 1\}$
- $3\text{-COLORING} \equiv$ CSP($\{\neq\}$) over $D = \{1, 2, 3\}$



Fixed-language CSPs

A **constraint language** Γ is a finite set of Boolean functions over a finite domain D

CSP(Γ) = CSP restricted to networks in which every constraint uses a function from Γ

- $\mathrm{3\text{-}SAT} \equiv$ CSP($\{$ternary Boolean clauses$\}$) over $D = \{0, 1\}$
- $\mathrm{3\text{-}COLORING} \equiv$ CSP($\{\neq\}$) over $D = \{1, 2, 3\}$
- etc.

What is the complexity of the following languages, over the domain $\{0, 1\}$?

- $\Gamma_0 = \{c_{\vee 3}\}$, where $c_{\vee 3}(x, y, z) = x \vee y \vee z$
- $\Gamma_1 = \{c_{\oplus}\}$, where $c_{\oplus}(x, y) = \{(0, 1), (1, 0)\}$
- $\Gamma_2 = \{c_{\vee 3}, c_{\oplus}\}$
- $\Gamma_3 = \{c_{\text{1-in-3}}\}$, where $c_{\text{1-in-3}}(x, y, z) = \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$

Fixed-language CSPs

A **constraint language** Γ is a finite set of Boolean functions over a finite domain D

CSP(Γ) $=$ CSP restricted to networks in which every constraint uses a function from Γ

- $3\text{-SAT} \equiv$ CSP({ternary Boolean clauses}) over $D = \{0, 1\}$
- $3\text{-COLORING} \equiv$ CSP($\{\neq\}$) over $D = \{1, 2, 3\}$
- etc.

What is the complexity of the following languages, over the domain $\{0, 1\}$?

- $\Gamma_0 = \{c_{\vee 3}\}$, where $c_{\vee 3}(x, y, z) = x \vee y \vee z$ **P**
- $\Gamma_1 = \{c_{\oplus}\}$, where $c_{\oplus}(x, y) = \{(0, 1), (1, 0)\}$
- $\Gamma_2 = \{c_{\vee 3}, c_{\oplus}\}$
- $\Gamma_3 = \{c_{\text{1-in-3}}\}$, where $c_{\text{1-in-3}}(x, y, z) = \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$

Fixed-language CSPs

A **constraint language** Γ is a finite set of Boolean functions over a finite domain D

CSP(Γ) = CSP restricted to networks in which every constraint uses a function from Γ

- $\text{3-SAT} \equiv$ CSP($\{$ternary Boolean clauses$\}$) over $D = \{0, 1\}$
- $\text{3-COLORING} \equiv$ CSP($\{\neq\}$) over $D = \{1, 2, 3\}$
- etc.

What is the complexity of the following languages, over the domain $\{0, 1\}$?

- $\Gamma_0 = \{c_{\vee 3}\}$, where $c_{\vee 3}(x, y, z) = x \vee y \vee z$ **P**
- $\Gamma_1 = \{c_{\oplus}\}$, where $c_{\oplus}(x, y) = \{(0, 1), (1, 0)\}$ **P**
- $\Gamma_2 = \{c_{\vee 3}, c_{\oplus}\}$
- $\Gamma_3 = \{c_{\text{1-in-3}}\}$, where $c_{\text{1-in-3}}(x, y, z) = \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$

Fixed-language CSPs

A **constraint language** Γ is a finite set of Boolean functions over a finite domain D

CSP(Γ) $=$ CSP restricted to networks in which every constraint uses a function from Γ

- $3\text{-SAT} \equiv \text{CSP}(\{\text{ternary Boolean clauses}\})$ over $D = \{0, 1\}$
- $3\text{-COLORING} \equiv \text{CSP}(\{\neq\})$ over $D = \{1, 2, 3\}$
- etc.

What is the complexity of the following languages, over the domain $\{0, 1\}$?

- $\Gamma_0 = \{c_{\vee 3}\}$, where $c_{\vee 3}(x, y, z) = x \vee y \vee z$ **P**
- $\Gamma_1 = \{c_{\oplus}\}$, where $c_{\oplus}(x, y) = \{(0, 1), (1, 0)\}$ **P**
- $\Gamma_2 = \{c_{\vee 3}, c_{\oplus}\}$ **NP-complete**
- $\Gamma_3 = \{c_{\text{1-in-3}}\}$, where $c_{\text{1-in-3}}(x, y, z) = \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$

Fixed-language CSPs

A **constraint language** Γ is a finite set of Boolean functions over a finite domain D

CSP(Γ) $=$ CSP restricted to networks in which every constraint uses a function from Γ

- $\mathrm{3\text{-}SAT} \equiv$ CSP($\{$ternary Boolean clauses$\}$) over $D = \{0,1\}$
- $\mathrm{3\text{-}COLORING} \equiv$ CSP($\{\neq\}$) over $D = \{1,2,3\}$
- etc.

What is the complexity of the following languages, over the domain $\{0,1\}$?

- $\Gamma_0 = \{c_{\vee 3}\}$, where $c_{\vee 3}(x,y,z) = x \vee y \vee z$ — **P**
- $\Gamma_1 = \{c_{\oplus}\}$, where $c_{\oplus}(x,y) = \{(0,1),(1,0)\}$ — **P**
- $\Gamma_2 = \{c_{\vee 3}, c_{\oplus}\}$ — **NP-complete**
- $\Gamma_3 = \{c_{\text{1-in-3}}\}$, where $c_{\text{1-in-3}}(x,y,z) = \{(1,0,0),(0,1,0),(0,0,1)\}$ — **NP-complete**

Gadgets

- Given a constraint language Γ, proving that CSP(Γ) is NP-hard is almost always done by finding an NP-hard language whose functions can be "expressed" by those in Γ
  - E.g. $x \neq y \equiv (x + z = y) \wedge (z \neq 0)$
  - May involve introducing new variables (as in decompositions)
  - These constructions are often called **gadgets**

- Gadgets can be very complicated, and sometimes difficult to craft

- A typical example: the textbook proof that CSP($\{c_{\text{1-in-3}}\}$) is NP-complete

CSP($\{c_{\text{1-in-3}}\}$) is NP-complete: Step 1

Step 1: $\{c_{\text{1-in-3}}\}$ can express $\{c_{\text{1-in-3}}, c_{\oplus}\}$

Let $N = (X, D, C)$ be a constraint network over $\{c_{\text{1-in-3}}, c_{\oplus}\}$.

For each constraint $c_{\oplus}(x, y)$, introduce two fresh variables z, w and replace $c_{\oplus}(x, y)$ with

$$c_{\text{1-in-3}}(z, z, w)$$
$$c_{\text{1-in-3}}(x, y, z)$$

CSP($\{c_{\text{1-in-3}}\}$) is NP-complete: Step 1

Step 1: $\{c_{\text{1-in-3}}\}$ can express $\{c_{\text{1-in-3}}, c_{\oplus}\}$

Let $N = (X, D, C)$ be a constraint network over $\{c_{\text{1-in-3}}, c_{\oplus}\}$.

For each constraint $c_{\oplus}(x, y)$, introduce two fresh variables z, w and replace $c_{\oplus}(x, y)$ with

$$c_{\text{1-in-3}}(z, z, w)$$
$$c_{\text{1-in-3}}(x, y, z)$$

Effect:

- $z = 0$, $w = 1$
- $(x, y) \in \{(0, 1), (1, 0)\} = c_{\oplus}$

CSP($\{c_{\text{1-in-3}}\}$) is NP-complete: Step 2

Step 2: $\{c_{\text{1-in-3}}, c_{\oplus}\}$ can express $\{c_{\vee 3}, c_{\oplus}\}$

Let $N = (X, D, C)$ be a constraint network over $\{c_{\vee 3}, c_{\oplus}\}$.

CSP($\{c_{\text{1-in-3}}\}$) is NP-complete: Step 2

Step 2: $\{c_{\text{1-in-3}}, c_\oplus\}$ can express $\{c_{\vee 3}, c_\oplus\}$

Let $N = (X, D, C)$ be a constraint network over $\{c_{\vee 3}, c_\oplus\}$.

For each constraint $c_{\vee 3}(x, y, z) \in C$, introduce six fresh variables $l_1, l_2, v_1, v_2, v_3, v_4$ and replace $c_{\vee 3}(x, y, z) \in C$ with

$$c_\oplus(x, l_1) \qquad c_{\text{1-in-3}}(l_1, v_1, v_2)$$
$$c_\oplus(z, l_2) \qquad c_{\text{1-in-3}}(y, v_2, v_3)$$
$$c_{\text{1-in-3}}(l_2, v_3, v_4)$$

CSP($\{c_{\text{1-in-3}}\}$) is NP-complete: Step 2

Step 2: $\{c_{\text{1-in-3}}, c_{\oplus}\}$ can express $\{c_{\vee 3}, c_{\oplus}\}$

Let $N = (X, D, C)$ be a constraint network over $\{c_{\vee 3}, c_{\oplus}\}$.

For each constraint $c_{\vee 3}(x, y, z) \in C$, introduce six fresh variables $l_1, l_2, v_1, v_2, v_3, v_4$ and replace $c_{\vee 3}(x, y, z) \in C$ with

$$
\begin{array}{lll}
 & \qquad\qquad & c_{\text{1-in-3}}(l_1, v_1, v_2) \\
c_{\oplus}(x, l_1) & & c_{\text{1-in-3}}(y, v_2, v_3) \\
c_{\oplus}(z, l_2) & & c_{\text{1-in-3}}(l_2, v_3, v_4)
\end{array}
$$

$c_{\vee 3}(x, y, z)$ satisfied $\Rightarrow$ new constraints satisfied:

- if $y = 1$: set $l_1 = \overline{x}$, $l_2 = \overline{z}$, $v_1 = x$, $v_2 = v_3 = 0$, $v_4 = z$
- else if $z = 1$: set $l_1 = \overline{x}$, $l_2 = 0$, $v_1 = x$, $v_2 = 0$, $v_3 = 1$, $v_4 = 0$
- else if $x = 1$: set $l_1 = 0$, $l_2 = 1$, $v_1 = 0$, $v_2 = 1$, $v_3 = 0$, $v_4 = 0$

CSP($\{c_{\text{1-in-3}}\}$) is NP-complete: Step 2

Step 2: $\{c_{\text{1-in-3}}, c_{\oplus}\}$ can express $\{c_{\vee 3}, c_{\oplus}\}$

Let $N = (X, D, C)$ be a constraint network over $\{c_{\vee 3}, c_{\oplus}\}$.

For each constraint $c_{\vee 3}(x, y, z) \in C$, introduce six fresh variables $l_1, l_2, v_1, v_2, v_3, v_4$ and replace $c_{\vee 3}(x, y, z) \in C$ with

$$
\begin{array}{ll}
 & c_{\text{1-in-3}}(l_1, v_1, v_2) \\
c_{\oplus}(x, l_1) & \\
 & c_{\text{1-in-3}}(y, v_2, v_3) \\
c_{\oplus}(z, l_2) & \\
 & c_{\text{1-in-3}}(l_2, v_3, v_4)
\end{array}
$$

$c_{\vee 3}(x, y, z)$ **not** satisfied $\Rightarrow$ new constraints **not** satisfied:

- $(x, y, z) = (0, 0, 0) \Rightarrow l_1 = 1,\ l_2 = 1,\ 1 \in \{v_2, v_3\}$: impossible

CSP($\{c_{\text{1-in-3}}\}$) is NP-complete: Step 2

Step 2: $\{c_{\text{1-in-3}}, c_{\oplus}\}$ can express $\{c_{\vee 3}, c_{\oplus}\}$

Let $N = (X, D, C)$ be a constraint network over $\{c_{\vee 3}, c_{\oplus}\}$.

For each constraint $c_{\vee 3}(x, y, z) \in C$, introduce six fresh variables $l_1, l_2, v_1, v_2, v_3, v_4$ and replace $c_{\vee 3}(x, y, z) \in C$ with

$$c_{\oplus}(x, l_1) \qquad c_{\text{1-in-3}}(l_1, v_1, v_2)$$
$$c_{\oplus}(z, l_2) \qquad c_{\text{1-in-3}}(y, v_2, v_3)$$
$$c_{\text{1-in-3}}(l_2, v_3, v_4)$$

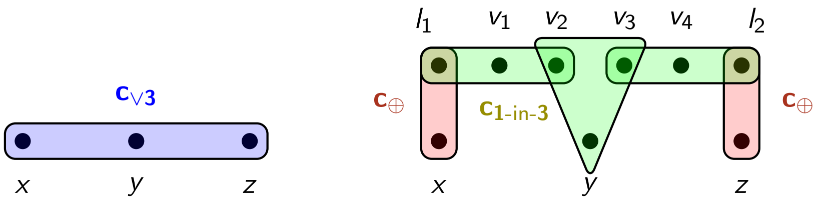
$c_{\vee 3}(x, y, z)$ **not** satisfied $\Rightarrow$ new constraints **not** satisfied:

- $(x, y, z) = (0, 0, 0) \Rightarrow l_1 = 1,\ l_2 = 1,\ 1 \in \{v_2, v_3\}$: impossible

Chaining Step 1 and Step 2 gives that $\{c_{\text{1-in-3}}\}$ can express $\{c_{\vee 3}, c_{\oplus}\}$, so CSP($\{c_{\text{1-in-3}}\}$) is **NP-complete**

Step 2 as a Boolean formula

$$x \vee y \vee z \iff \begin{aligned} &\exists l_1, l_2, v_1, v_2, v_3, v_4 : \\ &\quad c_{\oplus}(x, l_1) \wedge \\ &\quad c_{\oplus}(z, l_2) \wedge \\ &\quad c_{1\text{-in-}3}(l_1, v_1, v_2) \wedge \\ &\quad c_{1\text{-in-}3}(y, v_2, v_3) \wedge \\ &\quad c_{1\text{-in-}3}(l_2, v_3, v_4) \end{aligned}$$



pp-definitions

Definition

A **primitive-positive formula** (pp-formula) over a language Γ is a formula that only uses conjunction, existential quantification, equality and functions in Γ.

pp-definitions

Definition

A **primitive-positive formula** (pp-formula) over a language Γ is a formula that only uses conjunction, existential quantification, equality and functions in Γ.

Definition

A function c is **primitive-positive definable** (pp-definable) over a language Γ if there exists a pp-formula ϕ over Γ such that

$$(d_1, \ldots, d_r) \in c \iff \phi(d_1, \ldots, d_r) \text{ is true}$$

Expressivity

Theorem (folklore)

If every function $c \in \Gamma_1$ is pp-definable over Γ_2, then there is a (logspace) polynomial-time reduction from $\mathrm{CSP}(\Gamma_1)$ to $\mathrm{CSP}(\Gamma_2)$.

Intuition: If the function of a constraint c is pp-definable over Γ through a formula ϕ, then

- Add one fresh variable for each existentially quantified variable in ϕ
- Replace c with constraints from Γ given by the atoms of ϕ

Expressivity

Theorem (folklore)

If every function $c \in \Gamma_1$ is pp-definable over Γ_2, then there is a (logspace) polynomial-time reduction from $\mathrm{CSP}(\Gamma_1)$ to $\mathrm{CSP}(\Gamma_2)$.

Intuition: If the function of a constraint c is pp-definable over Γ through a formula ϕ, then

- Add one fresh variable for each existentially quantified variable in ϕ
- Replace c with constraints from Γ given by the atoms of ϕ

Definition

The **relational clone** of a language Γ, denoted by $\langle\Gamma\rangle$, is the set of all functions pp-definable over Γ.

Intuition: $\langle\Gamma\rangle \approx$ set of all functions that can be "expressed" by Γ

The complexity of CSP(Γ) is tightly connected to the functions in $\langle\Gamma\rangle$:

- Any algorithm that solves CSP(Γ) also solves CSP(Γ') for every $\Gamma' \subset \langle\Gamma\rangle$
- If $\langle\Gamma\rangle$ contains a well-known NP-hard language, then Γ is NP-hard
- If it does not, we can deduce useful structural information about Γ: the existence of non-trivial **polymorphisms**.

Polymorphisms

Given an operation $f : D^k \to D$ and k tuples $\tau_1, \dots, \tau_k \in D^q$, we write $f(\tau_1, \dots, \tau_k)$ for the tuple $(f(\tau_1[1], \dots, \tau_k[1]), \dots, f(\tau_1[q], \dots, \tau_k[q]))$

$$
\begin{array}{rl}
\tau_1 = & (\alpha_1 \quad \alpha_2 \quad \cdots \quad \alpha_q) \\
\tau_2 = & (\beta_1 \quad \beta_2 \quad \cdots \quad \beta_q) \\
\dots & \\
\tau_k = & (\gamma_1 \quad \gamma_2 \quad \cdots \quad \gamma_q)
\end{array}
$$

Polymorphisms

Given an operation $f : D^k \to D$ and k tuples $\tau_1, \dots, \tau_k \in D^q$, we write $f(\tau_1, \dots, \tau_k)$ for the tuple $(f(\tau_1[1], \dots, \tau_k[1]), \dots, f(\tau_1[q], \dots, \tau_k[q]))$

$$
\begin{array}{rcccc}
 & f & f & f & f \\
 & \frown & \frown & \frown & \frown \\
\tau_1 = & (\alpha_1 & \alpha_2 & \cdots & \alpha_q) \\
\tau_2 = & (\beta_1 & \beta_2 & \cdots & \beta_q) \\
\dots & & & & \\
\tau_k = & (\gamma_1 & \gamma_2 & \cdots & \gamma_q) \\
 & \smile & \smile & \smile & \smile \\
 & \parallel & \parallel & \parallel & \parallel \\
 & (\,\cdot\,, & \cdot\,, & \cdots, & \cdot\,)
\end{array}
$$

Polymorphisms

Given an operation $f : D^k \to D$ and k tuples $\tau_1, \dots, \tau_k \in D^q$, we write $f(\tau_1, \dots, \tau_k)$ for the tuple $(f(\tau_1[1], \dots, \tau_k[1]), \dots, f(\tau_1[q], \dots, \tau_k[q]))$

$$
\begin{array}{rccccc}
 & f & f & f & f \\
 & \frown & \frown & \frown & \frown \\
\tau_1 = & (\alpha_1 & \alpha_2 & \cdots & \alpha_q) \\
\tau_2 = & (\beta_1 & \beta_2 & \cdots & \beta_q) \\
\dots & & & & \\
\tau_k = & (\gamma_1 & \gamma_2 & \cdots & \gamma_q) \\
 & \smile & \smile & \smile & \smile \\
 & \| & \| & \| & \| \\
f(\tau_1, \dots, \tau_k) = & (\,\cdot\,, & \cdot\,, & \cdots\,, & \cdot\,)
\end{array}
$$

Polymorphisms

Given an operation $f : D^k \to D$ and k tuples $\tau_1, \ldots, \tau_k \in D^q$, we write $f(\tau_1, \ldots, \tau_k)$ for the tuple $(f(\tau_1[1], \ldots, \tau_k[1]), \ldots, f(\tau_1[q], \ldots, \tau_k[q]))$

Definition

Let Γ be over D. A k-ary **polymorphism** of Γ is an operation $f : D^k \to D$ such that $\forall c \in \Gamma$ and $\forall \tau_1, \ldots, \tau_k \in c$, we have that $f(\tau_1, \ldots, \tau_k) \in c$.

Polymorphisms

Given an operation $f : D^k \to D$ and k tuples $\tau_1, \ldots, \tau_k \in D^q$, we write $f(\tau_1, \ldots, \tau_k)$ for the tuple $(f(\tau_1[1], \ldots, \tau_k[1]), \ldots, f(\tau_1[q], \ldots, \tau_k[q]))$

Definition

Let Γ be over D. A k-ary **polymorphism** of Γ is an operation $f : D^k \to D$ such that $\forall c \in \Gamma$ and $\forall \tau_1, \ldots, \tau_k \in c$, we have that $f(\tau_1, \ldots, \tau_k) \in c$.

Example

$D = \{0, 1\}$, $f(a_1, a_2, a_3) = a_1 \oplus a_2 \oplus a_3 = \text{minority}(a_1, a_2, a_3)$

c :

1	1	1
1	0	1
0	1	0
0	0	0

Polymorphisms

Given an operation $f : D^k \to D$ and k tuples $\tau_1, \dots, \tau_k \in D^q$, we write $f(\tau_1, \dots, \tau_k)$ for the tuple $(f(\tau_1[1], \dots, \tau_k[1]), \dots, f(\tau_1[q], \dots, \tau_k[q]))$

Definition

Let Γ be over D. A k-ary **polymorphism** of Γ is an operation $f : D^k \to D$ such that $\forall c \in \Gamma$ and $\forall \tau_1, \dots, \tau_k \in c$, we have that $f(\tau_1, \dots, \tau_k) \in c$.

Example

$D = \{0, 1\}$, $f(a_1, a_2, a_3) = a_1 \oplus a_2 \oplus a_3 = \text{minority}(a_1, a_2, a_3)$

c :

1	1	1
1	0	1
0	1	0
0	0	0

Polymorphisms

Given an operation $f : D^k \to D$ and k tuples $\tau_1, \ldots, \tau_k \in D^q$, we write $f(\tau_1, \ldots, \tau_k)$ for the tuple $(f(\tau_1[1], \ldots, \tau_k[1]), \ldots, f(\tau_1[q], \ldots, \tau_k[q]))$

Definition

Let Γ be over D. A k-ary **polymorphism** of Γ is an operation $f : D^k \to D$ such that $\forall c \in \Gamma$ and $\forall \tau_1, \ldots, \tau_k \in c$, we have that $f(\tau_1, \ldots, \tau_k) \in c$.

Example

$D = \{0, 1\}$, $f(a_1, a_2, a_3) = a_1 \oplus a_2 \oplus a_3 = \text{minority}(a_1, a_2, a_3)$

$$c : \quad \begin{array}{|ccc|} \hline 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ \boxed{0} & 0 & 0 \\ \hline \end{array}$$

Polymorphisms

Given an operation $f : D^k \to D$ and k tuples $\tau_1, \ldots, \tau_k \in D^q$, we write $f(\tau_1, \ldots, \tau_k)$ for the tuple $(f(\tau_1[1], \ldots, \tau_k[1]), \ldots, f(\tau_1[q], \ldots, \tau_k[q]))$

Definition

Let Γ be over D. A k-ary **polymorphism** of Γ is an operation $f : D^k \to D$ such that $\forall c \in \Gamma$ and $\forall \tau_1, \ldots, \tau_k \in c$, we have that $f(\tau_1, \ldots, \tau_k) \in c$.

Example

$D = \{0, 1\}$, $f(a_1, a_2, a_3) = a_1 \oplus a_2 \oplus a_3 = \text{minority}(a_1, a_2, a_3)$

$$c : \quad \begin{array}{|ccc|} \hline 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & \boxed{0} & 0 \\ \hline \end{array}$$

Polymorphisms

Given an operation $f : D^k \to D$ and k tuples $\tau_1, \ldots, \tau_k \in D^q$, we write $f(\tau_1, \ldots, \tau_k)$ for the tuple $(f(\tau_1[1], \ldots, \tau_k[1]), \ldots, f(\tau_1[q], \ldots, \tau_k[q]))$

Definition

Let Γ be over D. A k-ary **polymorphism** of Γ is an operation $f : D^k \to D$ such that $\forall c \in \Gamma$ and $\forall \tau_1, \ldots, \tau_k \in c$, we have that $f(\tau_1, \ldots, \tau_k) \in c$.

Example

$D = \{0, 1\}$, $f(a_1, a_2, a_3) = a_1 \oplus a_2 \oplus a_3 = \text{minority}(a_1, a_2, a_3)$

$$c : \quad \begin{array}{|ccc|} \hline 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & \boxed{0} \\ \hline \end{array}$$

Polymorphisms

Given an operation $f : D^k \to D$ and k tuples $\tau_1, \ldots, \tau_k \in D^q$, we write $f(\tau_1, \ldots, \tau_k)$ for the tuple $(f(\tau_1[1], \ldots, \tau_k[1]), \ldots, f(\tau_1[q], \ldots, \tau_k[q]))$

Definition

Let Γ be over D. A k-ary **polymorphism** of Γ is an operation $f : D^k \to D$ such that $\forall c \in \Gamma$ and $\forall \tau_1, \ldots, \tau_k \in c$, we have that $f(\tau_1, \ldots, \tau_k) \in c$.

Example

$D = \{0, 1\}$, $f(a_1, a_2, a_3) = a_1 \oplus a_2 \oplus a_3 = \text{minority}(a_1, a_2, a_3)$

c :

1	1	1
1	0	1
0	1	0
0	0	0

Polymorphisms

Given an operation $f : D^k \to D$ and k tuples $\tau_1, \ldots, \tau_k \in D^q$, we write $f(\tau_1, \ldots, \tau_k)$ for the tuple $(f(\tau_1[1], \ldots, \tau_k[1]), \ldots, f(\tau_1[q], \ldots, \tau_k[q]))$

Definition

Let Γ be over D. A k-ary **polymorphism** of Γ is an operation $f : D^k \to D$ such that $\forall c \in \Gamma$ and $\forall \tau_1, \ldots, \tau_k \in c$, we have that $f(\tau_1, \ldots, \tau_k) \in c$.

Example

$D = \{0, 1\}$, $f(a_1, a_2, a_3) = a_1 \oplus a_2 \oplus a_3 = \text{minority}(a_1, a_2, a_3)$

$$c : \quad \begin{array}{|ccc|} \hline 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ \hline \end{array}$$

Polymorphisms

Notation: $\mathrm{Pol}(\Gamma)$ = set of polymorphisms of Γ

Proposition

Let Γ be a language over a domain D. Then $\mathrm{Pol}(\Gamma)$ is a *concrete clone*:

(i) $\mathrm{Pol}(\Gamma)$ **contains all projections**, i.e. operations $\pi_i : D^k \to D$ such that $\pi_i(a_1, \ldots, a_m) = a_i$.

(ii) $\mathrm{Pol}(\Gamma)$ is **closed under composition**, that is, any operation obtained by composing polymorphisms of Γ is also a polymorphism of Γ.

Example: if $f, g, h \in \mathrm{Pol}(\Gamma)$ are of respective arities $3, 2, 1$ then the operation $w : D^3 \to D$ given by

$$w(a_1, a_2, a_3) = f(f(a_1, a_1, h(a_2)), g(a_2, a_3), g(a_1, a_1))$$

is also a polymorphism of Γ.

Geiger's Theorem

Theorem [Geiger 1968]

Let Γ_1, Γ_2 be two constraint languages over the same domain D. Then, $\Gamma_2 \subseteq \langle \Gamma_1 \rangle$ if and only if $\mathrm{Pol}(\Gamma_1) \subseteq \mathrm{Pol}(\Gamma_2)$.

Consequence: the complexity of a language Γ is **completely determined** by its set of polymorphisms

Examples:

- $\mathrm{Pol}(\{c_\oplus\})$ contains the operation $f(a) = \overline{a}$, so no Boolean clause is pp-definable over $\{c_\oplus\}$
- $\mathrm{Pol}(\{c_{\text{1-in-3}}\})$ only contains projections, so every Boolean function is pp-definable over $\{c_{\text{1-in-3}}\}$

Polymorphisms and complexity

Intuition: If $\mathrm{Pol}(\Gamma)$ contains some highly nontrivial polymorphisms, then $\langle\Gamma\rangle$ is small, and Γ is unlikely to be NP-hard

Polymorphisms and complexity

Intuition: If $\mathrm{Pol}(\Gamma)$ contains some highly nontrivial polymorphisms, then $\langle\Gamma\rangle$ is small, and Γ is unlikely to be NP-hard

Some polymorphisms that guarantee polynomial-time solvability:

- Semilattice polymorphisms: $\forall a, b, c \in D$,

$$f(a,a) = a$$
$$f(a,b) = f(b,a)$$
$$f((f(a,b),c) = f(a,f(b,c))$$

- Majority polymorphisms: $f(a,a,b) = f(a,b,a) = f(b,a,a) = a$
- Minority polymorphisms: $f(a,a,b) = f(a,b,a) = f(b,a,a) = b$
- Mal'tsev polymorphisms: $f(a,a,b) = f(b,a,a) = b$
- Etc.

Example: Max(.,.)

Proposition

If Γ is a constraint language with polymorphism $f(a, b) = \max(a, b)$, then enforcing arc-consistency solves CSP(Γ).

Example: Max(.,.)

Proposition

If Γ is a constraint language with polymorphism $f(a, b) = \max(a, b)$, then enforcing arc-consistency solves CSP(Γ).

Claim 1

For any arity $k \geq 2$, the operation $\max(a_1, \ldots, a_k)$ is a polymorphism of Γ.

Example: Max(.,.)

Proposition

If Γ is a constraint language with polymorphism $f(a, b) = \max(a, b)$, then enforcing arc-consistency solves CSP(Γ).

Claim 1

For any arity $k \geq 2$, the operation $\max(a_1, \ldots, a_k)$ is a polymorphism of Γ.

Proof.

By induction on the arity k. True for $k = 2$. If true for $k - 1$, then

$$\max(a_1, \ldots, a_k) = \max(\max(a_1, \ldots, a_{k-1}), a_k)$$

is a polymorphism of Γ since Pol(Γ) is closed under composition. □

Example: Max(.,.)

Proposition

If Γ is a constraint language with polymorphism $f(a,b) = \max(a,b)$, then enforcing arc-consistency solves CSP(Γ).

Proof.

Let (X, D, C) be a network over Γ, and suppose that AC does not empty any domain.

For any constraint $c \in C$ with scheme $(x_1, \ldots, x_r)$ there exist r supports $\tau_1, \ldots, \tau_r \in c$ such that $\tau_i[x_i] = \max(D_{\text{AC}}(x_i))$.

By Claim 1, $\max(\tau_1, \ldots, \tau_r) = (\max(D_{\text{AC}}(x_1)), \ldots, \max(D_{\text{AC}}(x_r))) \in c$, so assigning each variable $x \in X$ to $\max(D_{\text{AC}}(x))$ satisfies all constraints. $\square$

Example: Max(.,.)

Proposition

If Γ is a constraint language with polymorphism $f(a, b) = \max(a, b)$, then enforcing arc-consistency solves CSP(Γ).

- Some max-closed functions:
  - $D \subset \mathbb{N}$, $c_>(x, y) = (x > y)$
  - $D = \{0, 1\}$, dual-Horn clauses:
    - $c(x, y_1, \ldots, y_q) = (y_1 \vee \ldots \vee y_q \vee \overline{x})$
    - $c(y_1, \ldots, y_q) = (y_1 \vee \ldots \vee y_q)$
- Also works for min (instead of max), e.g. Horn clauses:
  - $c(x, y_1, \ldots, y_q) = (\overline{y_1} \vee \ldots \vee \overline{y_q} \vee x)$
  - $c(y_1, \ldots, y_q) = (\overline{y_1} \vee \ldots \vee \overline{y_q})$

Schaefer's Theorem

Theorem [Schaefer 1978]

Let Γ be a finite Boolean constraint language. If Γ has one of the following polymorphisms:

- $f(a) = 0$
- $f(a) = 1$
- $f(a_1, a_2) = a_1 \vee a_2$
- $f(a_1, a_2) = a_1 \wedge a_2$
- $f(a_1, a_2, a_3) = \text{majority}(a_1, a_2, a_3)$
- $f(a_1, a_2, a_3) = \text{minority}(a_1, a_2, a_3)$

then CSP(Γ) is polynomial-time. Otherwise, CSP(Γ) is NP-complete.

Schaefer's Theorem

Theorem [Schaefer 1978]

Let Γ be a finite Boolean constraint language. If Γ has one of the following polymorphisms:

- $f(a) = 0$ [Return `true`]
- $f(a) = 1$ [Return `true`]
- $f(a_1, a_2) = a_1 \vee a_2$ [Enforce AC]
- $f(a_1, a_2) = a_1 \wedge a_2$ [Enforce AC]
- $f(a_1, a_2, a_3) = \text{majority}(a_1, a_2, a_3)$ [Enforce SAC]
- $f(a_1, a_2, a_3) = \text{minority}(a_1, a_2, a_3)$ [Gaussian elim.]

then CSP(Γ) is polynomial-time. Otherwise, CSP(Γ) is NP-complete.

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$$c_{\text{1-in-3}}(x, y, z) = \begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \end{array} & \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

- $f(a) = 0$:
- $f(a) = 1$:
- $f(a_1, a_2) = a_1 \vee a_2$:
- $f(a_1, a_2) = a_1 \wedge a_2$:
- $f(a_1, a_2, a_3) = \text{majority}(a_1, a_2, a_3)$:
- $f(a_1, a_2, a_3) = \text{minority}(a_1, a_2, a_3)$:

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$$c_{\text{1-in-3}}(x,y,z) = \begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \end{array} & \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

- $f(a) = 0$: **no** $\quad f(\tau_1) = (0,0,0) \notin c_{\text{1-in-3}}$
- $f(a) = 1$:
- $f(a_1, a_2) = a_1 \vee a_2$:
- $f(a_1, a_2) = a_1 \wedge a_2$:
- $f(a_1, a_2, a_3) = \text{majority}(a_1, a_2, a_3)$:
- $f(a_1, a_2, a_3) = \text{minority}(a_1, a_2, a_3)$:

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$$c_{\text{1-in-3}}(x,y,z) = \begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \end{array} & \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

- $f(a) = 0$: **no** — $f(\tau_1) = (0,0,0) \notin c_{\text{1-in-3}}$
- $f(a) = 1$: **no** — $f(\tau_1) = (1,1,1) \notin c_{\text{1-in-3}}$
- $f(a_1, a_2) = a_1 \vee a_2$:
- $f(a_1, a_2) = a_1 \wedge a_2$:
- $f(a_1, a_2, a_3) = \text{majority}(a_1, a_2, a_3)$:
- $f(a_1, a_2, a_3) = \text{minority}(a_1, a_2, a_3)$:

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$$c_{\text{1-in-3}}(x,y,z) = \begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \end{array} & \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

- $f(a) = 0$: **no** — $f(\tau_1) = (0,0,0) \notin c_{\text{1-in-3}}$
- $f(a) = 1$: **no** — $f(\tau_1) = (1,1,1) \notin c_{\text{1-in-3}}$
- $f(a_1,a_2) = a_1 \vee a_2$: **no** — $f(\tau_1,\tau_2) = (1,1,0) \notin c_{\text{1-in-3}}$
- $f(a_1,a_2) = a_1 \wedge a_2$:
- $f(a_1,a_2,a_3) = \text{majority}(a_1,a_2,a_3)$:
- $f(a_1,a_2,a_3) = \text{minority}(a_1,a_2,a_3)$:

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$$c_{\text{1-in-3}}(x,y,z) = \begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \end{array} & \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

- $f(a) = 0$: **no** — $f(\tau_1) = (0,0,0) \notin c_{\text{1-in-3}}$
- $f(a) = 1$: **no** — $f(\tau_1) = (1,1,1) \notin c_{\text{1-in-3}}$
- $f(a_1, a_2) = a_1 \vee a_2$: **no** — $f(\tau_1, \tau_2) = (1,1,0) \notin c_{\text{1-in-3}}$
- $f(a_1, a_2) = a_1 \wedge a_2$: **no** — $f(\tau_1, \tau_2) = (0,0,0) \notin c_{\text{1-in-3}}$
- $f(a_1, a_2, a_3) = \text{majority}(a_1, a_2, a_3)$:
- $f(a_1, a_2, a_3) = \text{minority}(a_1, a_2, a_3)$:

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$$c_{\text{1-in-3}}(x,y,z) = \begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \end{array} & \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

- $f(a) = 0$: **no** $\quad f(\tau_1) = (0,0,0) \notin c_{\text{1-in-3}}$
- $f(a) = 1$: **no** $\quad f(\tau_1) = (1,1,1) \notin c_{\text{1-in-3}}$
- $f(a_1,a_2) = a_1 \vee a_2$: **no** $\quad f(\tau_1,\tau_2) = (1,1,0) \notin c_{\text{1-in-3}}$
- $f(a_1,a_2) = a_1 \wedge a_2$: **no** $\quad f(\tau_1,\tau_2) = (0,0,0) \notin c_{\text{1-in-3}}$
- $f(a_1,a_2,a_3) = \text{majority}(a_1,a_2,a_3)$: **no** $\quad f(\tau_1,\tau_2,\tau_3) = (0,0,0) \notin c_{\text{1-in-3}}$
- $f(a_1,a_2,a_3) = \text{minority}(a_1,a_2,a_3)$:

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$$c_{\text{1-in-3}}(x,y,z) = \begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \end{array} & \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

- $f(a) = 0$: **no** $f(\tau_1) = (0,0,0) \notin c_{\text{1-in-3}}$
- $f(a) = 1$: **no** $f(\tau_1) = (1,1,1) \notin c_{\text{1-in-3}}$
- $f(a_1,a_2) = a_1 \vee a_2$: **no** $f(\tau_1,\tau_2) = (1,1,0) \notin c_{\text{1-in-3}}$
- $f(a_1,a_2) = a_1 \wedge a_2$: **no** $f(\tau_1,\tau_2) = (0,0,0) \notin c_{\text{1-in-3}}$
- $f(a_1,a_2,a_3) = \text{majority}(a_1,a_2,a_3)$: **no** $f(\tau_1,\tau_2,\tau_3) = (0,0,0) \notin c_{\text{1-in-3}}$
- $f(a_1,a_2,a_3) = \text{minority}(a_1,a_2,a_3)$: **no** $f(\tau_1,\tau_2,\tau_3) = (1,1,1) \notin c_{\text{1-in-3}}$

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$$c_{\text{1-in-3}}(x,y,z) = \begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \end{array} & \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \end{array}$$

NP-complete!

- $f(a) = 0$: **no** $f(\tau_1) = (0,0,0) \notin c_{\text{1-in-3}}$
- $f(a) = 1$: **no** $f(\tau_1) = (1,1,1) \notin c_{\text{1-in-3}}$
- $f(a_1,a_2) = a_1 \vee a_2$: **no** $f(\tau_1,\tau_2) = (1,1,0) \notin c_{\text{1-in-3}}$
- $f(a_1,a_2) = a_1 \wedge a_2$: **no** $f(\tau_1,\tau_2) = (0,0,0) \notin c_{\text{1-in-3}}$
- $f(a_1,a_2,a_3) = \text{majority}(a_1,a_2,a_3)$: **no** $f(\tau_1,\tau_2,\tau_3) = (0,0,0) \notin c_{\text{1-in-3}}$
- $f(a_1,a_2,a_3) = \text{minority}(a_1,a_2,a_3)$: **no** $f(\tau_1,\tau_2,\tau_3) = (1,1,1) \notin c_{\text{1-in-3}}$

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$$\Gamma = \{\text{all binary functions over } D = \{0, 1\}\}$$

- $f(a) = 0$:
- $f(a) = 1$:
- $f(a_1, a_2) = a_1 \vee a_2$:
- $f(a_1, a_2) = a_1 \wedge a_2$:
- $f(a_1, a_2, a_3) = \text{majority}(a_1, a_2, a_3)$:
- $f(a_1, a_2, a_3) = \text{minority}(a_1, a_2, a_3)$:

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$$\Gamma = \{\text{all binary functions over } D = \{0,1\}\}$$

- $f(a) = 0$: **no** $\{(1,1)\}$
- $f(a) = 1$:
- $f(a_1, a_2) = a_1 \vee a_2$:
- $f(a_1, a_2) = a_1 \wedge a_2$:
- $f(a_1, a_2, a_3) = \text{majority}(a_1, a_2, a_3)$:
- $f(a_1, a_2, a_3) = \text{minority}(a_1, a_2, a_3)$:

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$$\Gamma = \{\text{all binary functions over } D = \{0,1\}\}$$

- $f(a) = 0$: **no** $\{(1,1)\}$
- $f(a) = 1$: **no** $\{(0,0)\}$
- $f(a_1, a_2) = a_1 \vee a_2$:
- $f(a_1, a_2) = a_1 \wedge a_2$:
- $f(a_1, a_2, a_3) = \text{majority}(a_1, a_2, a_3)$:
- $f(a_1, a_2, a_3) = \text{minority}(a_1, a_2, a_3)$:

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$$\Gamma = \{\text{all binary functions over } D = \{0,1\}\}$$

- $f(a) = 0$: **no** $\{(1,1)\}$
- $f(a) = 1$: **no** $\{(0,0)\}$
- $f(a_1, a_2) = a_1 \vee a_2$: **no** $\{(0,1),(1,0)\}$
- $f(a_1, a_2) = a_1 \wedge a_2$:
- $f(a_1, a_2, a_3) = \text{majority}(a_1, a_2, a_3)$:
- $f(a_1, a_2, a_3) = \text{minority}(a_1, a_2, a_3)$:

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$$\Gamma = \{\text{all binary functions over } D = \{0,1\}\}$$

- $f(a) = 0$: **no** $\{(1,1)\}$
- $f(a) = 1$: **no** $\{(0,0)\}$
- $f(a_1, a_2) = a_1 \vee a_2$: **no** $\{(0,1),(1,0)\}$
- $f(a_1, a_2) = a_1 \wedge a_2$: **no** $\{(0,1),(1,0)\}$
- $f(a_1, a_2, a_3) = \text{majority}(a_1, a_2, a_3)$:
- $f(a_1, a_2, a_3) = \text{minority}(a_1, a_2, a_3)$:

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$$\Gamma = \{\text{all binary functions over } D = \{0,1\}\}$$

- $f(a) = 0$: **no** $\{(1,1)\}$
- $f(a) = 1$: **no** $\{(0,0)\}$
- $f(a_1, a_2) = a_1 \vee a_2$: **no** $\{(0,1),(1,0)\}$
- $f(a_1, a_2) = a_1 \wedge a_2$: **no** $\{(0,1),(1,0)\}$
- $f(a_1, a_2, a_3) = \text{majority}(a_1, a_2, a_3)$: **YES**
- $f(a_1, a_2, a_3) = \text{minority}(a_1, a_2, a_3)$:

Schaefer's Theorem

Schaefer's Theorem is often easy to use:

$\Gamma = \{$all binary functions over $D = \{0,1\}\}$ **PTIME!**

- $f(a) = 0$: **no** $\{(1,1)\}$
- $f(a) = 1$: **no** $\{(0,0)\}$
- $f(a_1, a_2) = a_1 \vee a_2$: **no** $\{(0,1),(1,0)\}$
- $f(a_1, a_2) = a_1 \wedge a_2$: **no** $\{(0,1),(1,0)\}$
- $f(a_1, a_2, a_3) = \text{majority}(a_1, a_2, a_3)$: **YES**
- $f(a_1, a_2, a_3) = \text{minority}(a_1, a_2, a_3)$: **no** $\{(0,1),(1,0),(1,1)\}$

The Dichotomy Theorem

Theorem [Bulatov 2017][Zhuk 2017]

Let Γ be a finite constraint language. If Γ has a polymorphism such that for all $a, r, e \in D$,

$$f(a, r, e, a) = f(r, a, r, e)$$

then CSP(Γ) is polynomial time. Otherwise, CSP(Γ) is NP-complete.

The Dichotomy Theorem

Theorem [Bulatov 2017][Zhuk 2017]

Let Γ be a finite constraint language. If Γ has a polymorphism such that for all $a, r, e \in D$,

$$f(a, r, e, a) = f(r, a, r, e)$$

then CSP(Γ) is polynomial time. Otherwise, CSP(Γ) is NP-complete.

- *Many* other equivalent formulations

The Dichotomy Theorem

Theorem [Bulatov 2017][Zhuk 2017]

Let Γ be a finite constraint language. If Γ has a polymorphism such that for all $a, r, e \in D$,

$$f(a, r, e, a) = f(r, a, r, e)$$

then CSP(Γ) is polynomial time. Otherwise, CSP(Γ) is NP-complete.

- *Many* other equivalent formulations
- There is a dichotomy: CSP(Γ) is either in P or NP-complete
  - Conjectured in 1993, proved 24 years later

The Dichotomy Theorem

Theorem [Bulatov 2017][Zhuk 2017]

Let Γ be a finite constraint language. If Γ has a polymorphism such that for all $a, r, e \in D$,

$$f(a, r, e, a) = f(r, a, r, e)$$

then CSP(Γ) is polynomial time. Otherwise, CSP(Γ) is NP-complete.

- *Many* other equivalent formulations
- There is a dichotomy: CSP(Γ) is either in P or NP-complete
  - Conjectured in 1993, proved 24 years later
- Generalises Schaefer's Theorem, but much harder to use
  - Checking if such a polymorphism exists is NP-complete

The Dichotomy Theorem

Theorem [Bulatov 2017][Zhuk 2017]

Let Γ be a finite constraint language. If Γ has a polymorphism such that for all $a, r, e \in D$,

$$f(a, r, e, a) = f(r, a, r, e)$$

then CSP(Γ) is polynomial time. Otherwise, CSP(Γ) is NP-complete.

- *Many* other equivalent formulations
- There is a dichotomy: CSP(Γ) is either in P or NP-complete
  - Conjectured in 1993, proved 24 years later
- Generalises Schaefer's Theorem, but much harder to use
  - Checking if such a polymorphism exists is NP-complete
- The polynomial-time algorithm is *extremely* impractical

The Dichotomy Theorem: Example 1 (easy)

$$\exists f \in \mathrm{Pol}(\Gamma) : \forall a, r, e \in D, f(a, r, e, a) = f(r, a, r, e)?$$

$$c_{\neq}(x, y) = \begin{array}{c c} & \begin{array}{cc} x & y \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \\ \tau_6 \end{array} & \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 1 \\ 2 & 3 \\ 3 & 1 \\ 3 & 2 \end{bmatrix} \end{array}$$

The Dichotomy Theorem: Example 1 (easy)

$$\exists f \in \mathrm{Pol}(\Gamma) : \forall a, r, e \in D, f(a, r, e, a) = f(r, a, r, e)?$$

$$c_{\neq}(x, y) = \begin{array}{c} \\ \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \\ \tau_6 \end{array} \begin{array}{c} \begin{array}{cc} x & y \end{array} \\ \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 1 \\ 2 & 3 \\ 3 & 1 \\ 3 & 2 \end{bmatrix} \end{array}$$

- Suppose that such a polymorphism f exists

The Dichotomy Theorem: Example 1 (easy)

$$\exists f \in \mathrm{Pol}(\Gamma) : \forall a, r, e \in D, f(a,r,e,a) = f(r,a,r,e)?$$

$$c_{\neq}(x,y) = \begin{array}{c c} & \begin{array}{cc} x & y \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \\ \tau_6 \end{array} & \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 1 \\ 2 & 3 \\ 3 & 1 \\ 3 & 2 \end{bmatrix} \end{array}$$

- Suppose that such a polymorphism f exists
- By the identity, $f(1,2,3,1) = f(2,1,2,3) = \alpha$ (α is unknown)

The Dichotomy Theorem: Example 1 (easy)

$$\exists f \in \mathrm{Pol}(\Gamma) : \forall a, r, e \in D, f(a, r, e, a) = f(r, a, r, e)?$$

$$c_{\neq}(x, y) = \begin{array}{c} \\ \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \\ \tau_6 \end{array} \begin{array}{c} \begin{array}{cc} x & y \end{array} \\ \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 1 \\ 2 & 3 \\ 3 & 1 \\ 3 & 2 \end{bmatrix} \end{array}$$

- Suppose that such a polymorphism f exists
- By the identity, $f(1, 2, 3, 1) = f(2, 1, 2, 3) = \alpha$ (α is unknown)
- But then $f(\tau_1, \tau_3, \tau_6, \tau_2) = (\alpha, \alpha) \in c_{\neq}$ **contradiction!**

The Dichotomy Theorem: Example 1 (easy)

$$\exists f \in \mathrm{Pol}(\Gamma) : \forall a, r, e \in D, f(a, r, e, a) = f(r, a, r, e)?$$

$$c_{\neq}(x, y) = \begin{array}{c} \\ \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \\ \tau_6 \end{array} \begin{array}{c} \begin{array}{cc} x & y \end{array} \\ \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 1 \\ 2 & 3 \\ 3 & 1 \\ 3 & 2 \end{bmatrix} \end{array}$$

NP-complete!

- Suppose that such a polymorphism f exists
- By the identity, $f(1, 2, 3, 1) = f(2, 1, 2, 3) = \alpha$ (α is unknown)
- But then $f(\tau_1, \tau_3, \tau_6, \tau_2) = (\alpha, \alpha) \in c_{\neq}$ **contradiction!**

The Dichotomy Theorem: Example 2 (a bit more difficult)

$$\exists f \in \mathrm{Pol}(\Gamma) : \forall a, r, e \in D, f(a, r, e, a) = f(r, a, r, e)?$$

$$c(x, y, z) = \begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \end{array} & \begin{bmatrix} 1 & 1 & 2 \\ 1 & 3 & 1 \\ 2 & 3 & 3 \\ 3 & 1 & 3 \\ 3 & 2 & 1 \end{bmatrix} \end{array}$$

The Dichotomy Theorem: Example 2 (a bit more difficult)

$$\exists f \in \mathrm{Pol}(\Gamma) : \forall a, r, e \in D, f(a, r, e, a) = f(r, a, r, e)?$$

$$c(x, y, z) = \begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \end{array} & \begin{bmatrix} 1 & 1 & 2 \\ 1 & 3 & 1 \\ 2 & 3 & 3 \\ 3 & 1 & 3 \\ 3 & 2 & 1 \end{bmatrix} \end{array}$$

- Suppose that such a polymorphism f exists

The Dichotomy Theorem: Example 2 (a bit more difficult)

$$\exists f \in \mathrm{Pol}(\Gamma) : \forall a, r, e \in D, f(a, r, e, a) = f(r, a, r, e)?$$

$$c(x, y, z) = \begin{array}{cc} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \end{array} & \begin{bmatrix} 1 & 1 & 2 \\ 1 & 3 & 1 \\ 2 & 3 & 3 \\ 3 & 1 & 3 \\ 3 & 2 & 1 \end{bmatrix} \end{array}$$

- Suppose that such a polymorphism f exists
- $f(\tau_4, \tau_2, \tau_1, \tau_4) \in c \Rightarrow f(3, 1, 1, 3) = f(1, 3, 1, 1) = 1, f(3, 1, 2, 3) = 2$

The Dichotomy Theorem: Example 2 (a bit more difficult)

$$\exists f \in \mathrm{Pol}(\Gamma) : \forall a, r, e \in D, f(a, r, e, a) = f(r, a, r, e)?$$

$$c(x, y, z) = \begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \end{array} & \begin{bmatrix} 1 & 1 & 2 \\ 1 & 3 & 1 \\ 2 & 3 & 3 \\ 3 & 1 & 3 \\ 3 & 2 & 1 \end{bmatrix} \end{array}$$

- Suppose that such a polymorphism f exists
- $f(\tau_4, \tau_2, \tau_1, \tau_4) \in c \Rightarrow f(3, 1, 1, 3) = f(1, 3, 1, 1) = 1, f(3, 1, 2, 3) = 2$
- From the identity: $f(1, 3, 1, 2) = 2$

The Dichotomy Theorem: Example 2 (a bit more difficult)

$$\exists f \in \mathrm{Pol}(\Gamma) : \forall a, r, e \in D, f(a, r, e, a) = f(r, a, r, e)?$$

$$c(x,y,z) = \begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \end{array} & \begin{bmatrix} 1 & 1 & 2 \\ 1 & 3 & 1 \\ 2 & 3 & 3 \\ 3 & 1 & 3 \\ 3 & 2 & 1 \end{bmatrix} \end{array}$$

- Suppose that such a polymorphism f exists
- $f(\tau_4, \tau_2, \tau_1, \tau_4) \in c \Rightarrow f(3,1,1,3) = f(1,3,1,1) = 1, f(3,1,2,3) = 2$
- From the identity: $f(1,3,1,2) = 2$
- $f(\tau_2, \tau_4, \tau_1, \tau_3) = (2, 1, ??) \in c$ **contradiction!**

The Dichotomy Theorem: Example 2 (a bit more difficult)

$$\exists f \in \mathrm{Pol}(\Gamma) : \forall a, r, e \in D, f(a, r, e, a) = f(r, a, r, e)?$$

$$c(x,y,z) = \begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \end{array} & \begin{bmatrix} 1 & 1 & 2 \\ 1 & 3 & 1 \\ 2 & 3 & 3 \\ 3 & 1 & 3 \\ 3 & 2 & 1 \end{bmatrix} \end{array}$$

NP-complete!

- Suppose that such a polymorphism f exists
- $f(\tau_4, \tau_2, \tau_1, \tau_4) \in c \Rightarrow f(3,1,1,3) = f(1,3,1,1) = 1, f(3,1,2,3) = 2$
- From the identity: $f(1,3,1,2) = 2$
- $f(\tau_2, \tau_4, \tau_1, \tau_3) = (2, 1, ??) \in c$ **contradiction!**

AC-solvable languages

Some polynomial-time languages Γ admit much simpler algorithms, e.g. arc consistency

AC-solvable languages

Some polynomial-time languages Γ admit much simpler algorithms, e.g. arc consistency

Theorem [Dalmau, Pearson 1999]

Let Γ be a constraint language. Then, CSP(Γ) is solved by AC if and only if for every $k > 0$ there exists a polymorphism f of arity k that is *totally symmetric*:

$$\text{If } \{a_1, \ldots, a_k\} = \{b_1, \ldots, b_k\}, \text{ then } f(a_1, \ldots, a_k) = f(b_1, \ldots, b_k)$$

Example: $\max(a_1, \ldots, a_k)$

Bounded width

A constraint language Γ has *bounded width* if there exists a finite k such that strong k-consistency solves CSP(Γ).

A *weak near-unanimity operation* (WNU) is an operation f such that $\forall a, b \in D,\ f(b, a, a, \ldots, a) = f(a, b, a, \ldots, a) = \ldots = f(a, a, a, \ldots, b)$.

Bounded width

A constraint language Γ has *bounded width* if there exists a finite k such that strong k-consistency solves CSP(Γ).

A *weak near-unanimity operation* (WNU) is an operation f such that $\forall a, b \in D,\ f(b, a, a, \ldots, a) = f(a, b, a, \ldots, a) = \ldots = f(a, a, a, \ldots, b)$.

Theorem [Barto, Kozik 2009]

Let Γ be a constraint language. Then, CSP(Γ) has bounded width if and only if it has two WNU polymorphisms f, g of arities 3, 4 s.t. $\forall a, b \in D$,

$$f(b, a, a) = g(b, a, a, a)$$

Bounded width

A constraint language Γ has *bounded width* if there exists a finite k such that strong k-consistency solves CSP(Γ).

A *weak near-unanimity operation* (WNU) is an operation f such that $\forall a, b \in D$, $f(b, a, a, \ldots, a) = f(a, b, a, \ldots, a) = \ldots = f(a, a, a, \ldots, b)$.

Theorem [Barto, Kozik 2009]

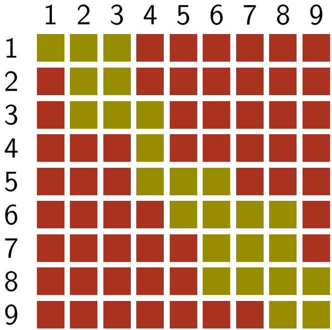
Let Γ be a constraint language. Then, CSP(Γ) has bounded width if and only if it has two WNU polymorphisms f, g of arities 3, 4 s.t. $\forall a, b \in D$,

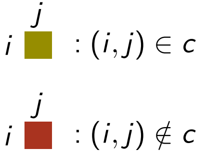
$$f(b, a, a) = g(b, a, a, a)$$

If Γ has bounded width, then SAC solves CSP(Γ)! [Kozik 2016]

Example

A binary constraint is *connected row-convex* if its compatibility matrix looks like this:

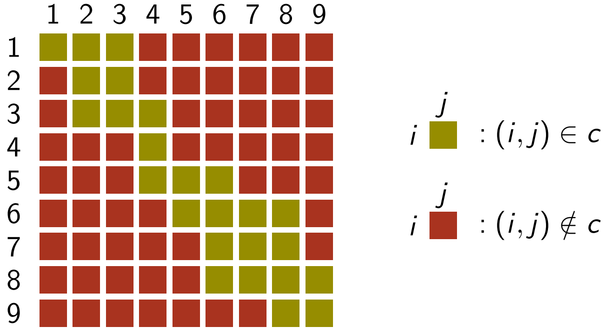




- row/columns = intervals, no disconnected parts
- e.g. $\alpha x + \beta y + c \geq 0$, $\alpha xy + b \geq 0$, limited accuracy measurements...

Example

A binary constraint is *connected row-convex* if its compatibility matrix looks like this:



- Polymorphism $f(a_1, a_2, a_3) = \text{median}(a_1, a_2, a_3)$
- Polymorphism $g(a_1, a_2, a_3, a_4) = f(f(a_1, a_2, a_3), a_3, a_4)$: SAC!

Other results

Polymorphisms can be used to obtain finer reductions, which preserve *exactly* the running times. For instance,

$$\begin{array}{cccccc} u & v & w & x & y & z \end{array}$$
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

is the *easiest* NP-complete language, but its exact complexity is unkown.

Polymorphisms can be extended to optimisation CSPs (constraints are cost functions) via *fractional polymorphisms*. For instance, on $D = \{0, 1\}$:

$$c(\tau_1) + c(\tau_2) \geq c(\tau_1 \wedge \tau_2) + c(\tau_1 \vee \tau_2)$$

Recap

- Fixed-language CSPs
- Reductions between languages: pp-definitions, clones
- Polymorphisms, Geiger's Theorem
- Schaefer's Theorem
- The Dichotomy Theorem
- Special cases: AC-solvable languages, bounded width

An excellent read: *Polymorphisms, and how to use them* [Barto, Krokhin, Willard 2017]

Part 2: structure-based tractable classes

Instance hypergraph

A *hypergraph* over a vertex set X is a collection of non-empty subsets of X called *edges*.

The *hypergraph of a constraint network* (X, D, C) has vertex set X, and its edges are given by the constraints' schemes:

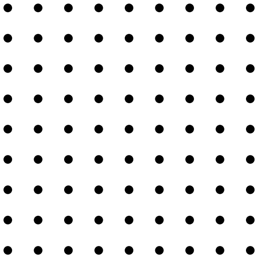
						6	8	
				7	3			9
3		9					4	5
4	9							
8		3		5		9		2
							3	6
9	6					3		8
7			6	8				
	2	8						

Instance hypergraph

A *hypergraph* over a vertex set X is a collection of non-empty subsets of X called *edges*.

The *hypergraph of a constraint network* (X, D, C) has vertex set X, and its edges are given by the constraints' schemes:

						6	8	
				7	3			9
3		9					4	5
4	9							
8		3		5		9		2
							3	6
9	6					3		8
7			6	8				
	2	8						

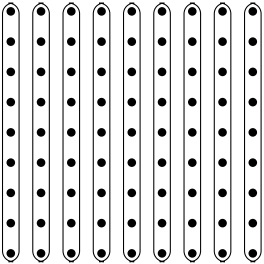


Instance hypergraph

A *hypergraph* over a vertex set X is a collection of non-empty subsets of X called *edges*.

The *hypergraph of a constraint network* (X, D, C) has vertex set X, and its edges are given by the constraints' schemes:

						6	8	
				7	3			9
3		9					4	5
4	9							
8		3		5		9		2
							3	6
9	6					3		8
7			6	8				
	2	8						

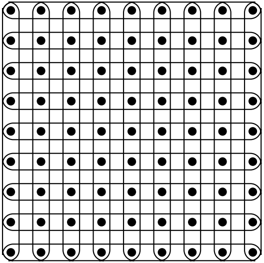


Instance hypergraph

A *hypergraph* over a vertex set X is a collection of non-empty subsets of X called *edges*.

The *hypergraph of a constraint network* (X, D, C) has vertex set X, and its edges are given by the constraints' schemes:

						6	8	
				7	3			9
3		9					4	5
4	9							
8		3		5		9		2
							3	6
9	6					3		8
7			6	8				
	2	8						

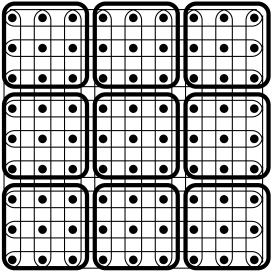


Instance hypergraph

A *hypergraph* over a vertex set X is a collection of non-empty subsets of X called *edges*.

The *hypergraph of a constraint network* (X, D, C) has vertex set X, and its edges are given by the constraints' schemes:

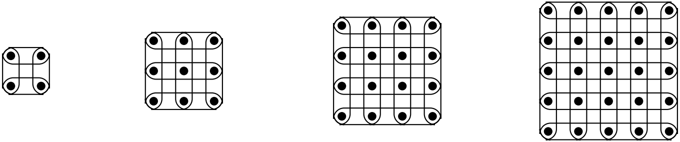
						6	8	
				7	3			9
3		9					4	5
4	9							
8		3		5		9		2
							3	6
9	6					3		8
7			6	8				
	2	8						



Structurally restricted CSP

If $\mathcal{H}$ is a family of hypergraphs, then CSP($\mathcal{H}$,-) is the CSP restricted to instances whose hypergraph is in $\mathcal{H}$

$\mathcal{H}$ is typically infinite:



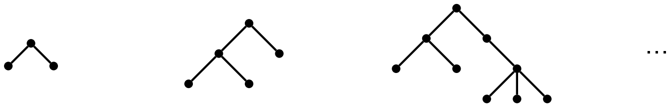
...

Note: $\mathcal{H}$ tells **nothing** about the constraints' functions!

Structurally restricted CSP

If $\mathcal{H}$ is a family of hypergraphs, then CSP($\mathcal{H}$,-) is the CSP restricted to instances whose hypergraph is in $\mathcal{H}$

$\mathcal{H}$ is typically infinite:



Note: $\mathcal{H}$ tells **nothing** about the constraints' functions!

Structurally restricted CSP

If $\mathcal{H}$ is a family of hypergraphs, then CSP($\mathcal{H}$,-) is the CSP restricted to instances whose hypergraph is in $\mathcal{H}$

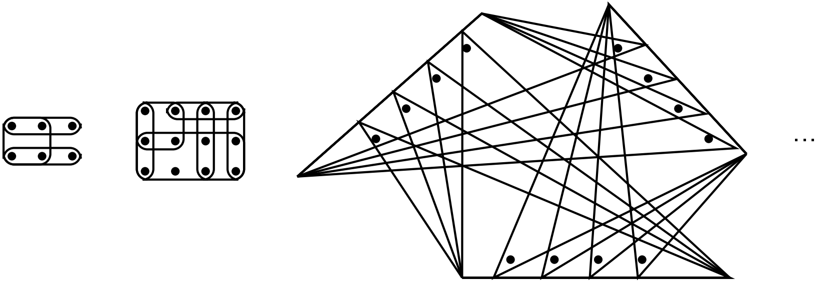
$\mathcal{H}$ is typically infinite:

Note: $\mathcal{H}$ tells **nothing** about the constraints' functions!

Structurally restricted CSP

If $\mathcal{H}$ is a family of hypergraphs, then CSP($\mathcal{H}$,-) is the CSP restricted to instances whose hypergraph is in $\mathcal{H}$

$\mathcal{H}$ is typically infinite:



Note: $\mathcal{H}$ tells **nothing** about the constraints' functions!

Structure vs language

For which families $\mathcal{H}$ is CSP($\mathcal{H}$,-) polynomial-time?

Key differences with language-based restrictions:

- Restricting to a fixed hypergraph does not make sense
  - Need to consider **infinite families**

Structure vs language

For which families $\mathcal{H}$ is CSP($\mathcal{H}$,-) polynomial-time?

Key differences with language-based restrictions:

- Restricting to a fixed hypergraph does not make sense
  - Need to consider **infinite families**
- Constraint arities may be unbounded
  - **Representation matters!** Recall that we assume a **positive table encoding** for all constraints

Structure vs language

For which families $\mathcal{H}$ is CSP($\mathcal{H}$,-) polynomial-time?

Key differences with language-based restrictions:

- Restricting to a fixed hypergraph does not make sense
  - Need to consider **infinite families**
- Constraint arities may be unbounded
  - **Representation matters!** Recall that we assume a **positive table encoding** for all constraints
- Full complexity classification is still an open problem

Structure vs language

For which families $\mathcal{H}$ is CSP($\mathcal{H}$,-) polynomial-time?

Key differences with language-based restrictions:

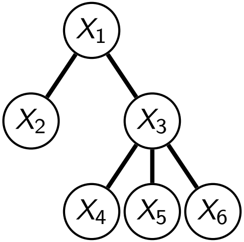
- Restricting to a fixed hypergraph does not make sense
  - Need to consider **infinite families**
- Constraint arities may be unbounded
  - **Representation matters!** Recall that we assume a **positive table encoding** for all constraints
- Full complexity classification is still an open problem
- Technical tools differ in nature: graph theory vs algebra

First observations: tree-structured CSPs

Consider $N \in \text{CSP}(\mathcal{H}\text{,-})$, where $\mathcal{H}$ **is the set of all tree graphs**.

Suppose that for every variable X, value $v \in D(X)$ and neighbour Y of X, there exists $v^* \in D(Y)$ such that $(X \leftarrow v, Y \leftarrow v^*)$ is consistent.

- Can be enforced in polytime, empty domain $\Rightarrow$ no solution

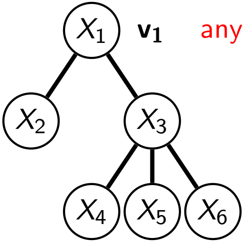


First observations: tree-structured CSPs

Consider $N \in \text{CSP}(\mathcal{H},\text{-})$, where $\mathcal{H}$ **is the set of all tree graphs**.

Suppose that for every variable X, value $v \in D(X)$ and neighbour Y of X, there exists $v^* \in D(Y)$ such that $(X \leftarrow v, Y \leftarrow v^*)$ is consistent.

- Can be enforced in polytime, empty domain $\Rightarrow$ no solution



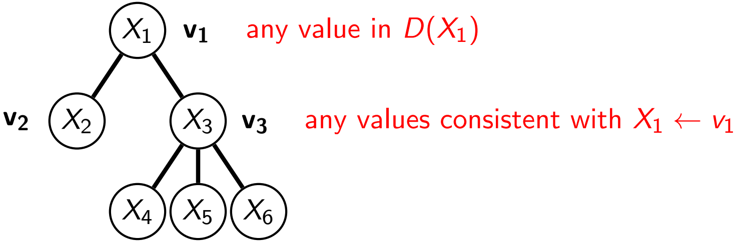
any value in $D(X_1)$

First observations: tree-structured CSPs

Consider $N \in \text{CSP}(\mathcal{H}\text{,-})$, where $\mathcal{H}$ **is the set of all tree graphs**.

Suppose that for every variable X, value $v \in D(X)$ and neighbour Y of X, there exists $v^* \in D(Y)$ such that $(X \leftarrow v, Y \leftarrow v^*)$ is consistent.

- Can be enforced in polytime, empty domain $\Rightarrow$ no solution

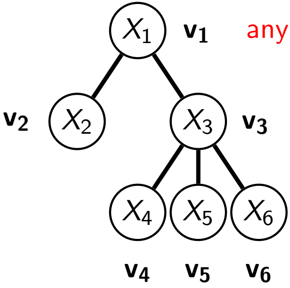


First observations: tree-structured CSPs

Consider $N \in \mathrm{CSP}(\mathcal{H}, \text{-})$, where $\mathcal{H}$ **is the set of all tree graphs**.

Suppose that for every variable X, value $v \in D(X)$ and neighbour Y of X, there exists $v^* \in D(Y)$ such that $(X \leftarrow v, Y \leftarrow v^*)$ is consistent.

- Can be enforced in polytime, empty domain $\Rightarrow$ no solution



any value in $D(X_1)$

any values consistent with $X_1 \leftarrow v_1$

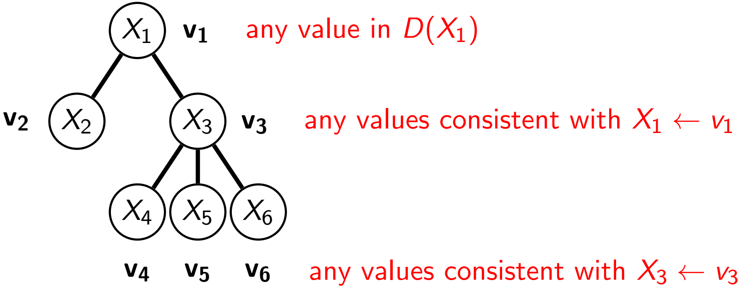
any values consistent with $X_3 \leftarrow v_3$

First observations: tree-structured CSPs

Consider $N \in \text{CSP}(\mathcal{H},\text{-})$, where $\mathcal{H}$ **is the set of all tree graphs**.

Suppose that for every variable X, value $v \in D(X)$ and neighbour Y of X, there exists $v^* \in D(Y)$ such that $(X \leftarrow v, Y \leftarrow v^*)$ is consistent.

- Can be enforced in polytime, empty domain $\Rightarrow$ no solution



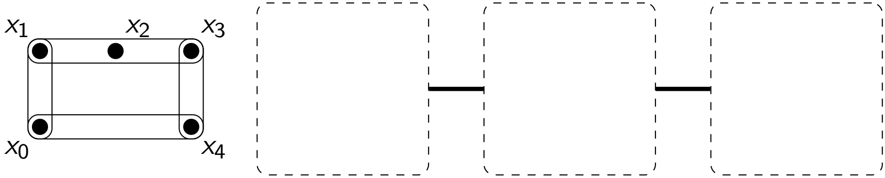
A solution always exists $\Rightarrow$ **CSP($\mathcal{H}$,-) is polynomial-time!**

Tree decompositions

Definition: Tree decomposition

A **tree decomposition** of a hypergraph H is a pair $(T, (B_t)_{t \in V(T)})$ such that

(i) T is a tree;



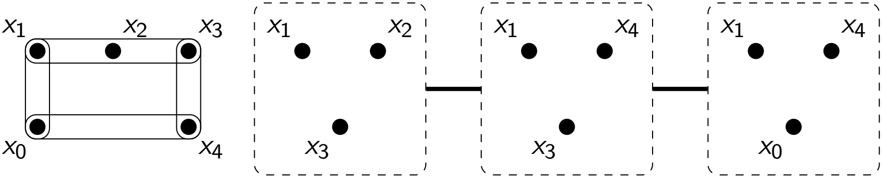
Tree decompositions

Definition: Tree decomposition

A **tree decomposition** of a hypergraph H is a pair $(T, (B_t)_{t \in V(T)})$ such that

(i) T is a tree;

(ii) Each "bag" B_t, $t \in V(T)$ is a set of vertices of H;



Tree decompositions

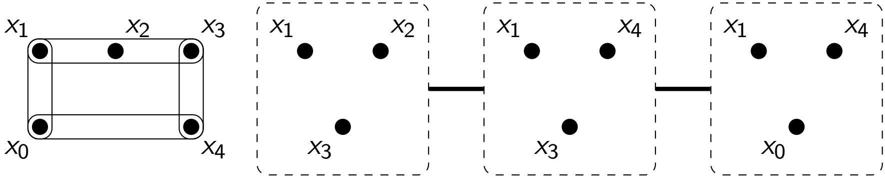
Definition: Tree decomposition

A **tree decomposition** of a hypergraph H is a pair $(T, (B_t)_{t \in V(T)})$ such that

(i) T is a tree;

(ii) Each "bag" B_t, $t \in V(T)$ is a set of vertices of H;

(iii) Each edge of H is **completely contained** in at least one bag;



Tree decompositions

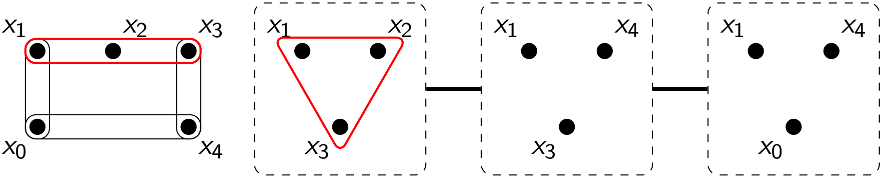
Definition: Tree decomposition

A **tree decomposition** of a hypergraph H is a pair $(T, (B_t)_{t \in V(T)})$ such that

(i) T is a tree;

(ii) Each "bag" B_t, $t \in V(T)$ is a set of vertices of H;

(iii) Each edge of H is **completely contained** in at least one bag;



Tree decompositions

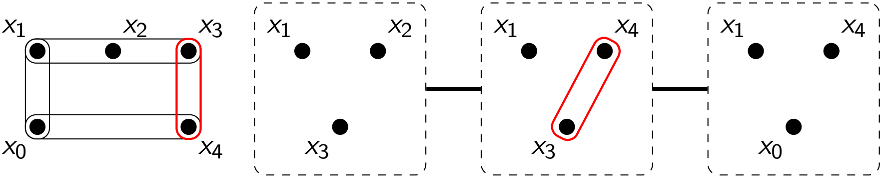
Definition: Tree decomposition

A **tree decomposition** of a hypergraph H is a pair $(T, (B_t)_{t \in V(T)})$ such that

(i) T is a tree;

(ii) Each "bag" B_t, $t \in V(T)$ is a set of vertices of H;

(iii) Each edge of H is **completely contained** in at least one bag;



Tree decompositions

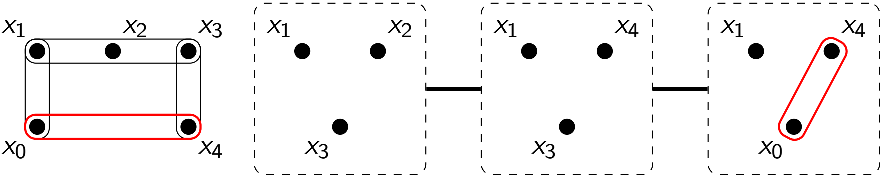
Definition: Tree decomposition

A **tree decomposition** of a hypergraph H is a pair $(T, (B_t)_{t \in V(T)})$ such that

(i) T is a tree;

(ii) Each "bag" B_t, $t \in V(T)$ is a set of vertices of H;

(iii) Each edge of H is **completely contained** in at least one bag;



Tree decompositions

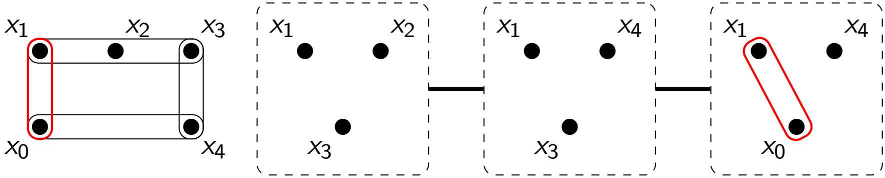
Definition: Tree decomposition

A **tree decomposition** of a hypergraph H is a pair $(T, (B_t)_{t \in V(T)})$ such that

(i) T is a tree;

(ii) Each "bag" B_t, $t \in V(T)$ is a set of vertices of H;

(iii) Each edge of H is **completely contained** in at least one bag;



Tree decompositions

Definition: Tree decomposition

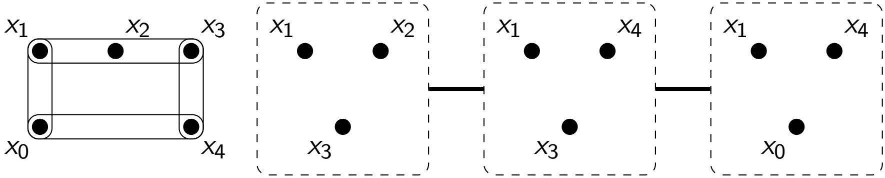
A **tree decomposition** of a hypergraph H is a pair $(T, (B_t)_{t \in V(T)})$ such that

(i) T is a tree;

(ii) Each "bag" B_t, $t \in V(T)$ is a set of vertices of H;

(iii) Each edge of H is **completely contained** in at least one bag;

(iv) For each vertex v of H, the bags that contain v form a **connected subtree** of T.



Tree decompositions

Definition: Tree decomposition

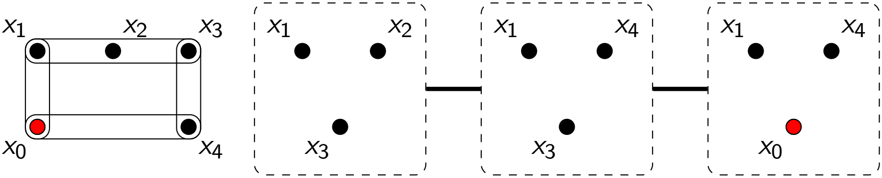
A **tree decomposition** of a hypergraph H is a pair $(T, (B_t)_{t \in V(T)})$ such that

(i) T is a tree;

(ii) Each "bag" B_t, $t \in V(T)$ is a set of vertices of H;

(iii) Each edge of H is **completely contained** in at least one bag;

(iv) For each vertex v of H, the bags that contain v form a **connected subtree** of T.



Tree decompositions

Definition: Tree decomposition

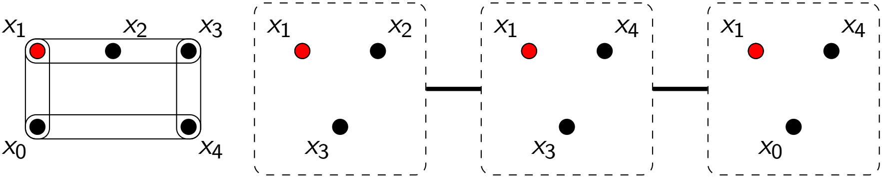
A **tree decomposition** of a hypergraph H is a pair $(T, (B_t)_{t \in V(T)})$ such that

(i) T is a tree;

(ii) Each "bag" B_t, $t \in V(T)$ is a set of vertices of H;

(iii) Each edge of H is **completely contained** in at least one bag;

(iv) For each vertex v of H, the bags that contain v form a **connected subtree** of T.



Tree decompositions

Definition: Tree decomposition

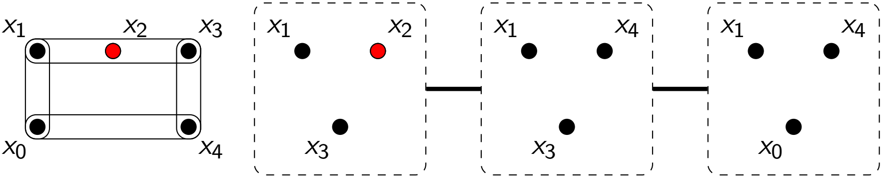
A **tree decomposition** of a hypergraph H is a pair $(T, (B_t)_{t \in V(T)})$ such that

(i) T is a tree;

(ii) Each "bag" B_t, $t \in V(T)$ is a set of vertices of H;

(iii) Each edge of H is **completely contained** in at least one bag;

(iv) For each vertex v of H, the bags that contain v form a **connected subtree** of T.



Tree decompositions

Definition: Tree decomposition

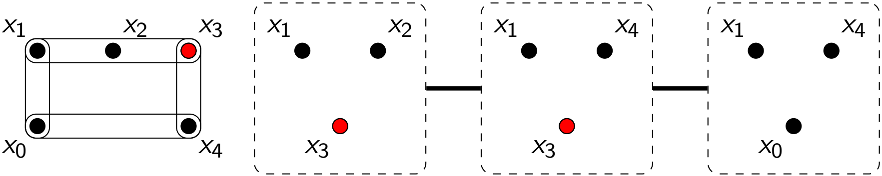
A **tree decomposition** of a hypergraph H is a pair $(T, (B_t)_{t \in V(T)})$ such that

(i) T is a tree;

(ii) Each "bag" B_t, $t \in V(T)$ is a set of vertices of H;

(iii) Each edge of H is **completely contained** in at least one bag;

(iv) For each vertex v of H, the bags that contain v form a **connected subtree** of T.



Tree decompositions

Definition: Tree decomposition

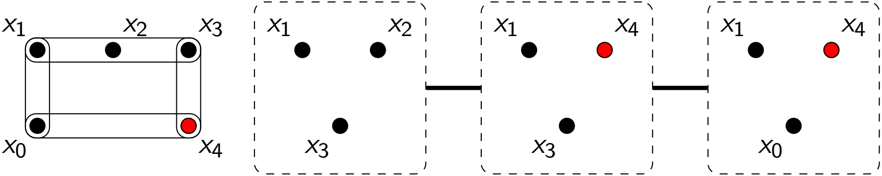
A **tree decomposition** of a hypergraph H is a pair $(T, (B_t)_{t \in V(T)})$ such that

(i) T is a tree;

(ii) Each "bag" B_t, $t \in V(T)$ is a set of vertices of H;

(iii) Each edge of H is **completely contained** in at least one bag;

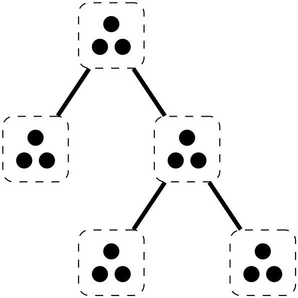
(iv) For each vertex v of H, the bags that contain v form a **connected subtree** of T.



Solving via tree decompositions

Fact

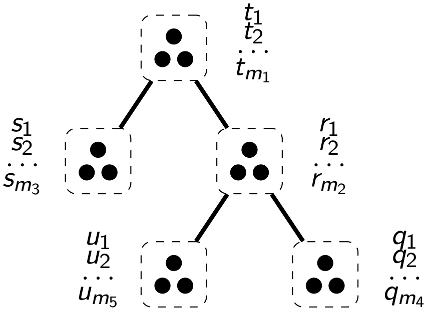
Given a CSP instance I and a tree decomposition of $H(I)$, **if we can enumerate all possible assignments for all bags in polynomial time, then we can solve I in polynomial time.**



Solving via tree decompositions

Fact

Given a CSP instance I and a tree decomposition of $H(I)$, **if we can enumerate all possible assignments for all bags in polynomial time, then we can solve I in polynomial time.**

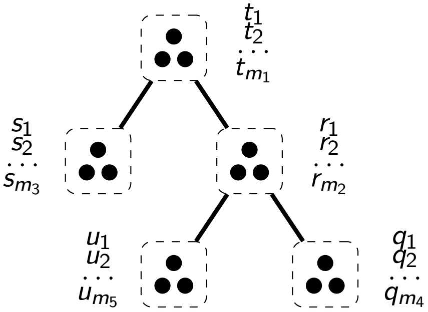


1. Compute all consistent assignments to bags

Solving via tree decompositions

Fact

Given a CSP instance I and a tree decomposition of $H(I)$, **if we can enumerate all possible assignments for all bags in polynomial time, then we can solve I in polynomial time.**

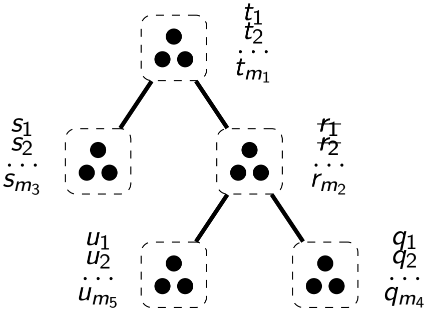


1. Compute all consistent assignments to bags
2. Bottom-up: remove assignments that cannot be extended to every child

Solving via tree decompositions

Fact

Given a CSP instance I and a tree decomposition of $H(I)$, **if we can enumerate all possible assignments for all bags in polynomial time, then we can solve I in polynomial time.**

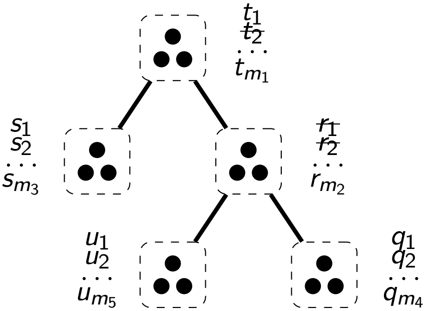


1. Compute all consistent assignments to bags
2. Bottom-up: remove assignments that cannot be extended to every child

Solving via tree decompositions

Fact

Given a CSP instance I and a tree decomposition of $H(I)$, **if we can enumerate all possible assignments for all bags in polynomial time, then we can solve I in polynomial time.**

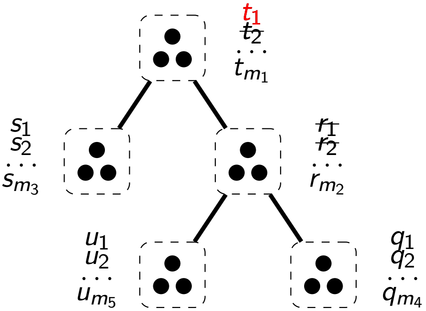


1. Compute all consistent assignments to bags
2. Bottom-up: remove assignments that cannot be extended to every child

Solving via tree decompositions

Fact

Given a CSP instance I and a tree decomposition of $H(I)$, **if we can enumerate all possible assignments for all bags in polynomial time, then we can solve I in polynomial time.**

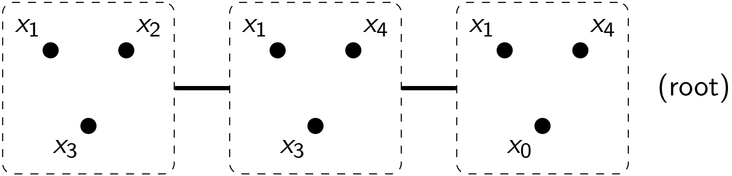


1. Compute all consistent assignments to bags
2. Bottom-up: remove assignments that cannot be extended to every child
3. Return YES iff there is at least one assignment left at the root

Example

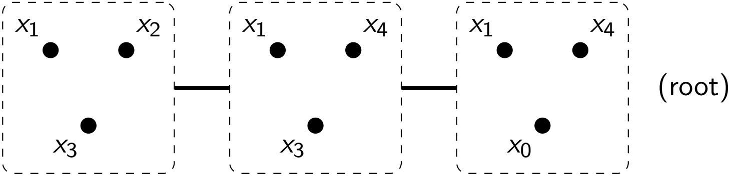
$D = \{0,1\}$, constraints:

$$c_{\text{1-in-3}}(x_1, x_2, x_3)$$
$$x_2 \vee x_3$$
$$x_3 \vee x_4$$
$$c_{\oplus}(x_4, x_0)$$
$$x_0 \vee x_1$$



Example

$D = \{0, 1\}$, constraints:
$$c_{\text{1-in-3}}(x_1, x_2, x_3)$$
$$x_2 \vee x_3$$
$$x_3 \vee x_4$$
$$c_{\oplus}(x_4, x_0)$$
$$x_0 \vee x_1$$



$$\begin{array}{ccc} x_1 & x_2 & x_3 \\ \end{array}$$
$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

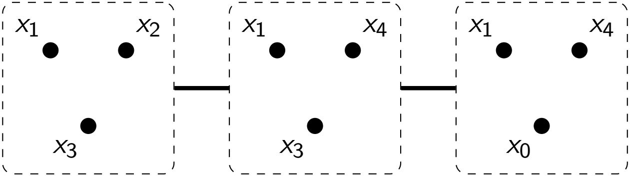
$$\begin{array}{ccc} x_1 & x_4 & x_3 \\ \end{array}$$
$$\begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{array}{ccc} x_1 & x_4 & x_0 \\ \end{array}$$
$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Example

$D = \{0, 1\}$, constraints:

$$c_{1\text{-in-}3}(x_1, x_2, x_3)$$
$$x_2 \vee x_3$$
$$x_3 \vee x_4$$
$$c_{\oplus}(x_4, x_0)$$
$$x_0 \vee x_1$$



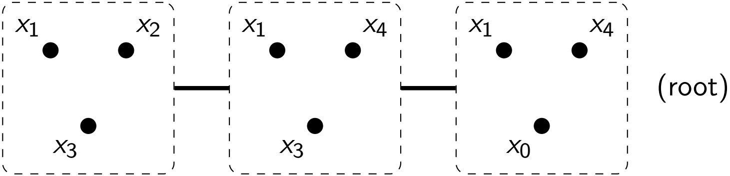
(root)

$$\begin{array}{c} \begin{array}{ccc} x_1 & x_2 & x_3 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array} \qquad \begin{array}{c} \begin{array}{ccc} x_1 & x_4 & x_3 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ \cancel{1} & \cancel{1} & \cancel{0} \\ 0 & 1 & 0 \end{bmatrix} \end{array} \qquad \begin{array}{c} \begin{array}{ccc} x_1 & x_4 & x_0 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \end{array}$$

Example

$D = \{0, 1\}$, constraints:

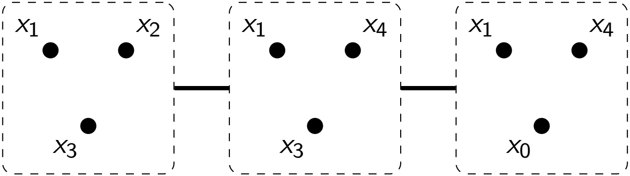
$$c_{1\text{-in-}3}(x_1, x_2, x_3)$$
$$x_2 \vee x_3$$
$$x_3 \vee x_4$$
$$c_{\oplus}(x_4, x_0)$$
$$x_0 \vee x_1$$



$$\begin{array}{c} \begin{array}{ccc} x_1 & x_2 & x_3 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array} \qquad \begin{array}{c} \begin{array}{ccc} x_1 & x_4 & x_3 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ \cancel{1} & \cancel{1} & \cancel{0} \\ 0 & 1 & 0 \end{bmatrix} \end{array} \qquad \begin{array}{c} \begin{array}{ccc} x_1 & x_4 & x_0 \end{array} \\ \begin{bmatrix} \cancel{1} & \cancel{1} & \cancel{0} \\ 0 & 0 & 1 \\ \cancel{1} & \cancel{0} & \cancel{1} \end{bmatrix} \end{array}$$

Example

$$D = \{0,1\}, \text{ constraints: } \quad \begin{array}{c} c_{1\text{-in-}3}(x_1, x_2, x_3) \\ x_2 \vee x_3 \\ x_3 \vee x_4 \\ c_{\oplus}(x_4, x_0) \\ x_0 \vee x_1 \end{array}$$



(root)

$$\begin{array}{ccc} x_1 & x_2 & x_3 \end{array} \quad \begin{array}{ccc} x_1 & x_4 & x_3 \end{array} \quad \begin{array}{ccc} x_1 & x_4 & x_0 \end{array}$$

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ \cancel{1} & \cancel{1} & \cancel{0} \\ 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} \cancel{1} & \cancel{1} & \cancel{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \cancel{1} & \cancel{0} & \cancel{1} \end{bmatrix}$$

SAT!

Solving via tree decompositions

Algorithm Solve(N : CSP ; $(T, (B_t)_{t \in V(T)})$ tree dec. of $H(N)$):

for each $t \in V(T)$ **from the leaves to the root do**
  $S_t \leftarrow \{\tau : B_t \to D \mid \tau \text{ is consistent}\}$[1];
  for each $t_c \in V(T)$: t_c is a child of t **do**
    $S_t \leftarrow \{\tau \in S_t \mid \exists \tau_c \in S_{t_c} : \tau_c \cup \tau \text{ is consistent}\}$;
$r \leftarrow \text{root}(T)$;
return $(S_r \neq \emptyset)$;

Correctness:

- N has a solution $\phi \Rightarrow \phi[B_t] \in S_t$ for all $t \in V(T)$: Solve returns `true`
- Solve returns `true` $\Rightarrow$ root(T) has a consistent assignment $\phi_r \in S_r$
  - $\Rightarrow$ ϕ_r can be extended to a complete assignment ϕ by picking arbitrary compatible assignments in each S_t (uses vertex-connectivity)
  - $\Rightarrow$ ϕ satisfies all constraints (uses edge containement)

[1] here "consistent" means that it satisfies the projection onto B_t of every constraint.

Treewidth

The **width** of $(T, (B_t)_{t \in V(T)})$ is the maximum size of a bag B_t, minus one.

The **treewidth** of a hypergraph H is the minimum width of a tree decomposition of H.

Treewidth

The **width** of $(T, (B_t)_{t \in V(T)})$ is the maximum size of a bag B_t, minus one.

The **treewidth** of a hypergraph H is the minimum width of a tree decomposition of H.

Fact [Freuder 1990]

CSP($\mathcal{H}$,-) is polynomial-time if $\mathcal{H}$ has bounded treewidth.

Proof: Compute an optimal tree decomposition in linear time [Bodlaender 1992] and then use algorithm `Solve`.

Treewidth

The **width** of $(T, (B_t)_{t \in V(T)})$ is the maximum size of a bag B_t, minus one.

The **treewidth** of a hypergraph H is the minimum width of a tree decomposition of H.

Fact [Freuder 1990]

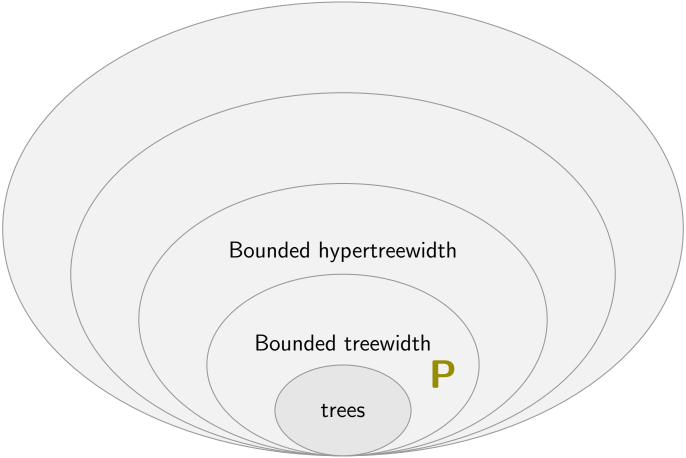
CSP($\mathcal{H}$,-) is polynomial-time if $\mathcal{H}$ has bounded treewidth.

Proof: Compute an optimal tree decomposition in linear time [Bodlaender 1992] and then use algorithm `Solve`.

Remarks:

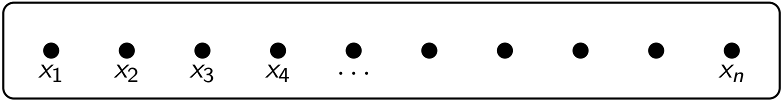
- Simply establishing strong $(k+1)$-consistency also works, where k is the treewidth of $\mathcal{H}$
- Under mild assumptions, if the arity is bounded then CSP($\mathcal{H}$,-) polynomial-time $\iff$ $\mathcal{H}$ has bounded treewidth [Grohe 2006]

Bounded hypertreewidth



Hypertreewidth

Problem: some **trivial** tractable hypergraph families have unbounded treewidth

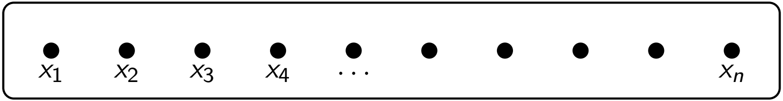


n variables, one constraint: $c(x_1, \ldots, x_n)$

treewidth is $n - 1$

Hypertreewidth

Problem: some **trivial** tractable hypergraph families have unbounded treewidth



n variables, one constraint: $c(x_1, \dots, x_n)$

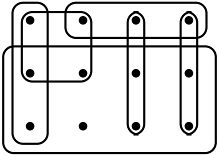
treewidth is $n - 1$

Solution: relax the width measure for tree decompositions

Hypertreewidth

Definition

Let H be an hypergraph with vertex set X. A subset $X' \subseteq X$ is *k-covered* iff there exist k edges $e_1, \ldots, e_k \in H$ such that $X' \subseteq e_1 \cup \ldots \cup e_k$.

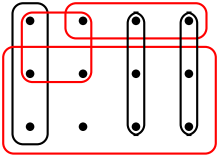


3-covered, but not 2-covered

Hypertreewidth

Definition

Let H be an hypergraph with vertex set X. A subset $X' \subseteq X$ is *k-covered* iff there exist k edges $e_1, \ldots, e_k \in H$ such that $X' \subseteq e_1 \cup \ldots \cup e_k$.



3-covered, but not 2-covered

Hypertreewidth

Fact

If a subset X' of variables is covered by k constraint with at most t tuples each, then it has at most t^k consistent assignments.

Proof sketch:

Let $c_1, \ldots, c_k$ be k covering constraints and $S = \{(\tau_1, \ldots, \tau_k) \mid \tau_i \in c_i\}$. Each consistent assignment ϕ to X' satisfies $c_1[X'], \ldots, c_k[X']$, so we can map ϕ to at least one element of S. Since $c_1, \ldots, c_k$ covers X', this mapping is injective: there are at most $|S| \leq t^k$ consistent assignments.

Hypertreewidth

The **c-width** of a tree decomposition $(T, (B_t)_{t \in V(T)})$ is the least k such that every bag B_t is k-covered

The (generalised) **hypertreewidth** of a hypergraph H is the minimum c-width of a tree decomposition of H [Gottlob, Leone, Scarcello 1999]

Hypertreewidth

The **c-width** of a tree decomposition $(T, (B_t)_{t \in V(T)})$ is the least k such that every bag B_t is k-covered

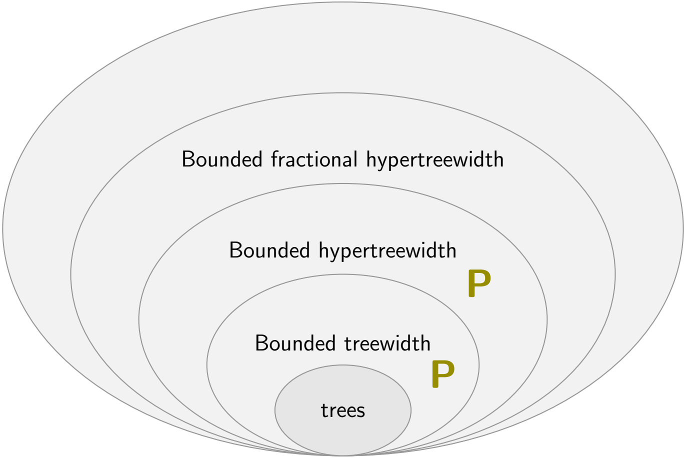
The (generalised) **hypertreewidth** of a hypergraph H is the minimum c-width of a tree decomposition of H [Gottlob, Leone, Scarcello 1999]

Theorem [Gottlob, Leone, Scarcello 1999]

CSP($\mathcal{H}$,-) is polynomial-time if $\mathcal{H}$ has bounded hypertreewidth.

- Given a fixed $k \geq 2$, computing a tree decomposition of c-width $\leq k$ if one exists is NP-hard [Fischl, Gottlob, Pichler 2018]
- It is however possible to compute a tree dec. of c-width at most $3k$ in polynomial time (or conclude that H has hypertreewidth $> k$)

Bounded fractional hypertreewidth



Bounded fractional hypertreewidth

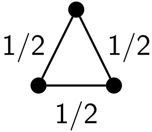
The algorithm `Solve` runs in polynomial time if

1. Each bag has **polynomially many consistent assignments**
2. They can be enumerated in **polynomial time**

Being covered by k constraints is a **sufficient** condition for that... but is it necessary?

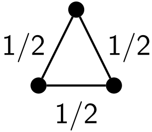
Bounded fractional hypertreewidth

A **fractional edge cover** of a hypergraph is a weight assignment γ to edges such that for every vertex v, $\sum_{e \in H : v \in E} \gamma(e) \geq 1$.



Bounded fractional hypertreewidth

A **fractional edge cover** of a hypergraph is a weight assignment γ to edges such that for every vertex v, $\sum_{e \in H : v \in E} \gamma(e) \geq 1$.



The AGM bound [Atserias, Grohe, Marx 2008]

If $I = (X, D, C)$ is a CSP instance with hypergraph H_I and γ is a fractional edge cover of H_I of total weight k, then I has at most

$$\Pi_{c \in C} |c|^{\gamma(c)} \leq t^k$$

solutions, and this upper bound is tight.

Fractional edge covers: an example

Consider $\mathcal{H} = \{H_n, n \in \mathbb{N}\}$, where H_n has

- one vertex v_S for each subset S of size n of $\{1, \ldots, 2n\}$, and
- one edge $e_i = \{v_S : i \in S\}$ for each $i \in \{1, \ldots, 2n\}$.

$$H_2 \quad = \quad \begin{array}{ccccc} \{1,2\} & & \{1,3\} & & \{1,4\} \\ & \{2,3\} & & \{3,4\} & \\ & & \{2,4\} & & \end{array}$$

Fractional edge covers: an example

Consider $\mathcal{H} = \{H_n, n \in \mathbb{N}\}$, where H_n has

- one vertex v_S for each subset S of size n of $\{1, \dots, 2n\}$, and
- one edge $e_i = \{v_S : i \in S\}$ for each $i \in \{1, \dots, 2n\}$.

$H_2 =$

$\{1,2\}$ $\{1,3\}$ $\{1,4\}$

$\{2,3\}$ $\{3,4\}$

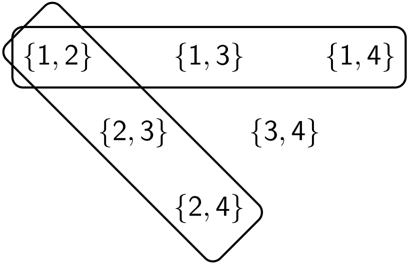
$\{2,4\}$

Fractional edge covers: an example

Consider $\mathcal{H} = \{H_n, n \in \mathbb{N}\}$, where H_n has

- one vertex v_S for each subset S of size n of $\{1, \ldots, 2n\}$, and
- one edge $e_i = \{v_S : i \in S\}$ for each $i \in \{1, \ldots, 2n\}$.

$H_2 \quad =$

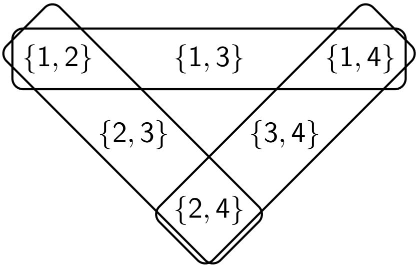


Fractional edge covers: an example

Consider $\mathcal{H} = \{H_n, n \in \mathbb{N}\}$, where H_n has

- one vertex v_S for each subset S of size n of $\{1, \ldots, 2n\}$, and
- one edge $e_i = \{v_S : i \in S\}$ for each $i \in \{1, \ldots, 2n\}$.

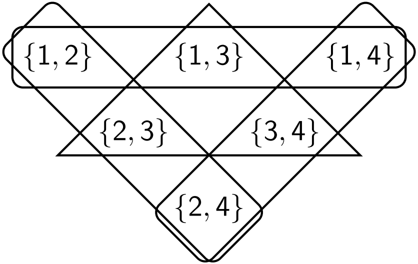
$H_2 =$



Fractional edge covers: an example

Consider $\mathcal{H} = \{H_n, n \in \mathbb{N}\}$, where H_n has

- one vertex v_S for each subset S of size n of $\{1, \ldots, 2n\}$, and
- one edge $e_i = \{v_S : i \in S\}$ for each $i \in \{1, \ldots, 2n\}$.

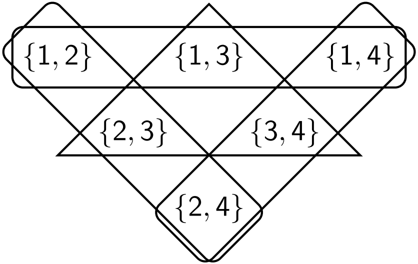
$H_2 =$ 

Fractional edge covers: an example

Consider $\mathcal{H} = \{H_n, n \in \mathbb{N}\}$, where H_n has

- one vertex v_S for each subset S of size n of $\{1, \dots, 2n\}$, and
- one edge $e_i = \{v_S : i \in S\}$ for each $i \in \{1, \dots, 2n\}$.

$H_2 \;=$



- For each n, the smallest edge cover of H_n has $n+1$ edges, but
- For each n, the smallest **fractional** edge cover of H_n has weight 2

Fractional hypertreewidth

The **fc-width** of $(T, (B_t)_{t \in V(T)})$ is the least k such that every bag B_t has a **fractional edge cover of weight at most** k.

The **fractional hypertreewidth** of H is the minimum fc-width of a tree decomposition of H [Grohe, Marx 2006].

Fractional hypertreewidth

The **fc-width** of $(T, (B_t)_{t \in V(T)})$ is the least k such that every bag B_t has a **fractional edge cover of weight at most** k.

The **fractional hypertreewidth** of H is the minimum fc-width of a tree decomposition of H [Grohe, Marx 2006].

Theorem [Grohe, Marx 2006]

CSP($\mathcal{H}$,-) is polynomial-time if $\mathcal{H}$ has bounded fractional hypertreewidth.

- As for hypertreewidth, given a fixed $k \geq 2$ deciding if there exists a tree decomposition of fc-width $\leq k$ is NP-hard
- An approximately optimal tree decomposition can be computed in polynomial time, but this time the best approximation known is k^3 (instead of $3k$ for hypertreewidth) [Marx 2009]

Treewidth vs Hypertreewidth vs Frac. Hypertreewidth

Consider the following (generic) planning problem:

- 4 objects disposed in a line, across timesteps $t = 1, \dots, n$.
- The state of an object at time $t+1$ depends on its state at time t, the state of adjacent objects at time t, and the action taken between t and $t+1$.

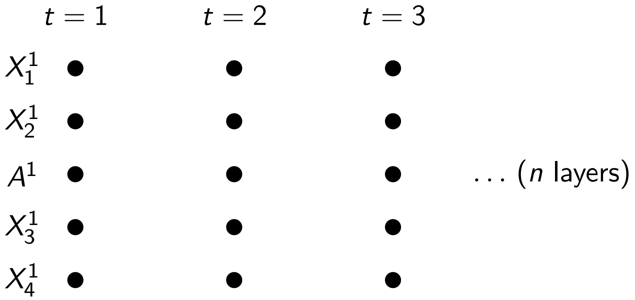
As a CSP:

- X_i^t = state of object i at time t, A^t = action between t and $t+1$
- constraints:

$$
\begin{aligned}
X_1^{t+1} &= f_1(X_1^t, X_2^t, A^t) && \forall t < n \\
X_2^{t+1} &= f_2(X_1^t, X_2^t, X_3^t, A^t) && \forall t < n \\
X_3^{t+1} &= f_3(X_2^t, X_3^t, X_4^t, A^t) && \forall t < n \\
X_4^{t+1} &= f_4(X_3^t, X_4^t, A^t) && \forall t < n
\end{aligned}
$$

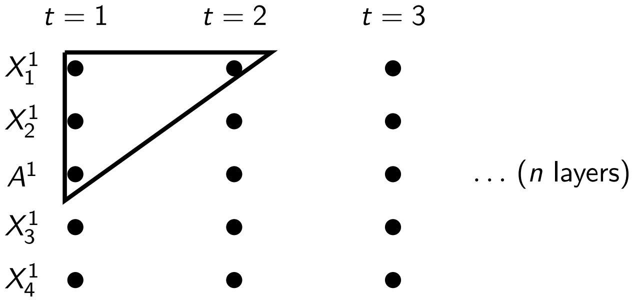
Treewidth vs Hypertreewidth vs Frac. Hypertreewidth

The hypergraph H of the network looks like this:



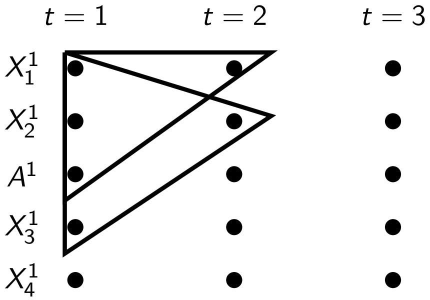
Treewidth vs Hypertreewidth vs Frac. Hypertreewidth

The hypergraph H of the network looks like this:



Treewidth vs Hypertreewidth vs Frac. Hypertreewidth

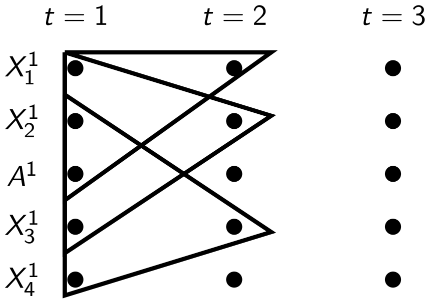
The hypergraph H of the network looks like this:



... (n layers)

Treewidth vs Hypertreewidth vs Frac. Hypertreewidth

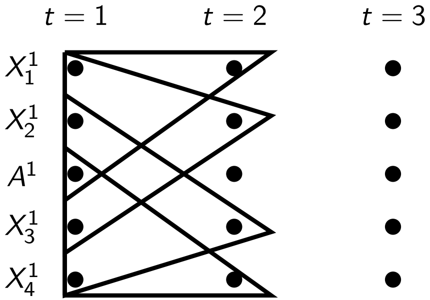
The hypergraph H of the network looks like this:



... (n layers)

Treewidth vs Hypertreewidth vs Frac. Hypertreewidth

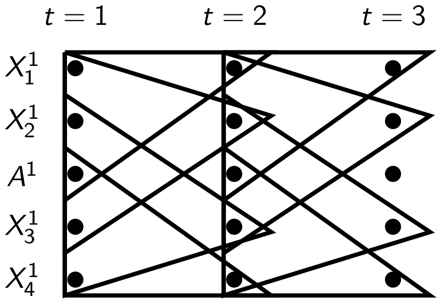
The hypergraph H of the network looks like this:



. . . (n layers)

Treewidth vs Hypertreewidth vs Frac. Hypertreewidth

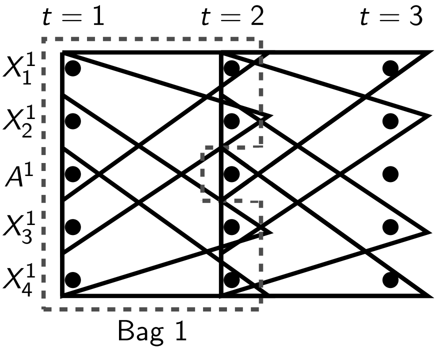
The hypergraph H of the network looks like this:



... (n layers)

Treewidth vs Hypertreewidth vs Frac. Hypertreewidth

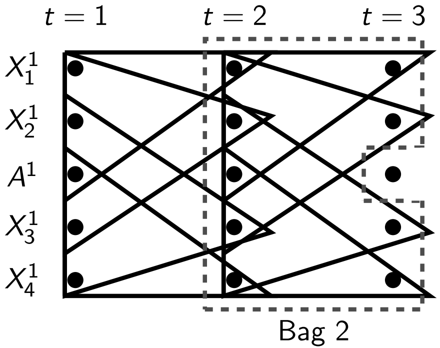
The hypergraph H of the network looks like this:



... (n layers)

Treewidth vs Hypertreewidth vs Frac. Hypertreewidth

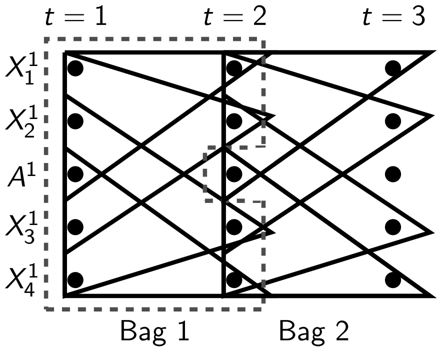
The hypergraph H of the network looks like this:



... (n layers)

Treewidth vs Hypertreewidth vs Frac. Hypertreewidth

The hypergraph H of the network looks like this:

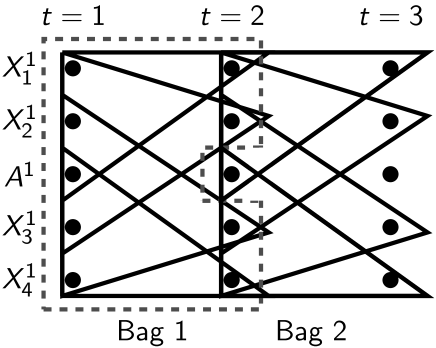


... (n layers)

... (T is a path + 4 leaves)

Treewidth vs Hypertreewidth vs Frac. Hypertreewidth

The hypergraph H of the network looks like this:



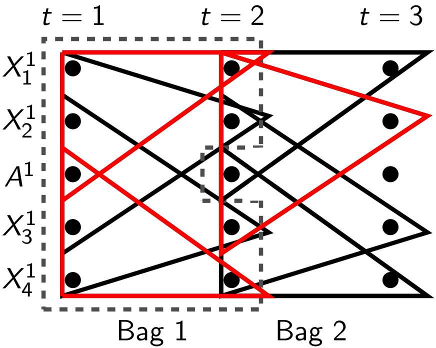
... (n layers)

... (T is a path + 4 leaves)

- width $= 7 \Rightarrow$ treewidth$(H) \leq 7$

Treewidth vs Hypertreewidth vs Frac. Hypertreewidth

The hypergraph H of the network looks like this:



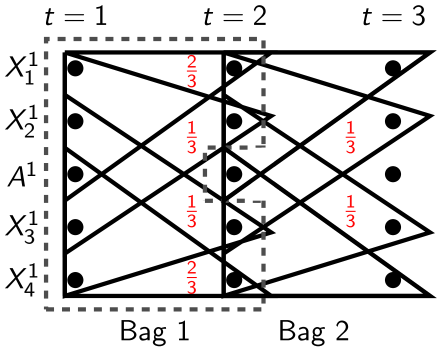
… (n layers)

… (T is a path + 4 leaves)

- width $= 7 \Rightarrow$ treewidth$(H) \leq 7$
- c-width $= 3 \Rightarrow$ hypertreewidth$(H) \leq 3$

Treewidth vs Hypertreewidth vs Frac. Hypertreewidth

The hypergraph H of the network looks like this:

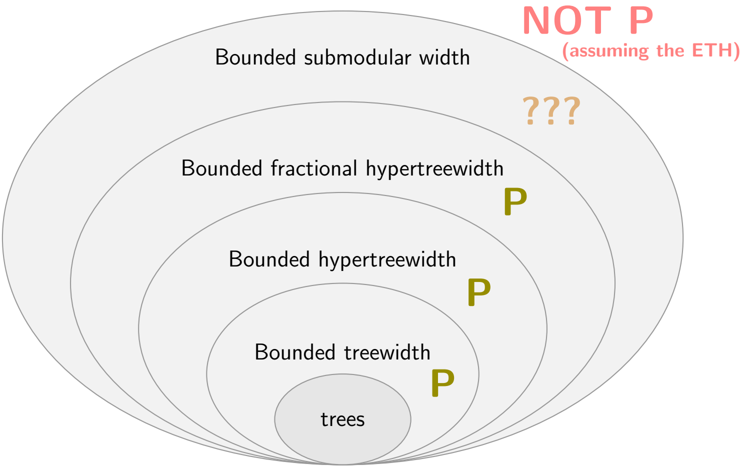


… (n layers)

… (T is a path + 4 leaves)

- width $= 7 \Rightarrow$ treewidth$(H) \leq 7$
- c-width $= 3 \Rightarrow$ hypertreewidth$(H) \leq 3$
- fc-width $= 8/3 \Rightarrow$ frac-hypertreewidth$(H) \leq 8/3$

State of the art: structure-based classes



References

[Geiger 1968] David Geiger. Closed systems of functions and predicates. Pacific J. Math., 1968.
[Schaefer 1978] Thomas J. Schaefer. The complexity of satisfiability problems. STOC'78.
[Bulatov 2017] Andrei A. Bulatov. A dichotomy theorem for nonuniform CSPs. FOCS'17.
[Zhuk 2017] Dmitriy Zhuk. A proof of CSP dichotomy conjecture. FOCS'17.
[Dalmau, Pearson 1999] Closure functions and width 1 problems. CP'99.
[Barto, Kozik 2009] Constraint satisfaction problems of bounded width. FOCS'09.
[Barto, Krokhin, Willard 2017] Polymorphisms, and how to use them. The Constraint Satisfaction Problem: Complexity and Approximability, Chapter 1. 2017.
[Freuder 1990] Complexity of k-tree structured constraint satisfaction problems. AAAI'90.
[Gottlob, Leone, Scarcello 1999] Hypertree decompositions and tractable queries. PODS'99.
[Fischl, Gottlob, Pichler 2018] General and fractional hypertree decompositions: hard and easy Cases. PODS'18.
[Atserias, Grohe, Marx 2008] Size bounds and query plans for relational joins. FOCS'08.
[Grohe, Marx 2006] Constraint solving via fractional edge covers. SODA'06.
[Marx 2009] Approximating fractional hypertree width. SODA'09.