Backdoors for SAT and CSP

Robert Ganian

PCSS 2017 · September 3, 2017

Overview

- This talk is about:
  - The **Boolean Satisfiability Problem** (SAT)
  - The **Constraint Satisfaction Problem** (CSP)
  - Fixed-parameter tractability
- This talk is not about:
  - Parameterizing by solution size
  - Kernelization
  - Model counting

SAT

- Input: a CNF formula *F*, for instance:

$$(x \lor y) \land (\neg x \lor z \lor y) \land (\neg y \lor \neg z)$$

- Terminology:
  - **variables** (3 – x, y, z)
  - **clauses** (3 – $(x \lor y)$, $(\neg x \lor z \lor y)$,$(\neg y \lor \neg z)$)
  - **literals** (7 – x, y, $\neg x$…)
- Question: Is *F* satisfiable?
  - Can you assign variables to 0/1 so that each clause is satisfied?

SAT

- Input: a CNF formula *F*, for instance:

$$(x \vee y) \wedge (\neg x \vee z \vee y) \wedge (\neg y \vee \neg z)$$

- Terminology:
  - **variables** (3 – x, y, z)
  - **clauses** (3 – $(x \vee y)$, $(\neg x \vee z \vee y)$,$(\neg y \vee \neg z)$)
  - **literals** (7 – x, y, $\neg x$)
- Question: Is *F* satisfiable?
  - Can you assign variables to 0/1 so that each clause is satisfied?
    - Example: x, y = 1, z = 0

SAT

- Input: a CNF formula *F*, for instance:

$$(1 \vee 1) \wedge (0 \vee 0 \vee 1) \wedge (0 \vee 1)$$

- Terminology:
  - **variables** (3 – x, y, z)
  - **clauses** (3 – $(x \vee y)$, $(\neg x \vee z \vee y)$,$(\neg y \vee \neg z)$)
  - **literals** (7 – x, y, $\neg x$)
- Question: Is *F* satisfiable?
  - Can you assign variables to 0/1 so that each clause is satisfied?
    - Example: x, y = 1, z = 0

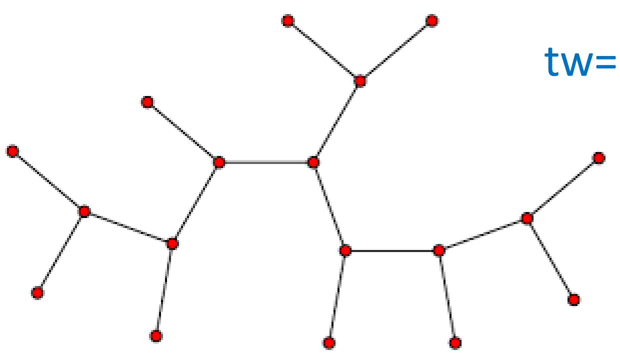
SAT

- **Many** applications
- One of the best known **NP**-complete problems
- Dedicated annual conference (SAT)
  - Also includes a SAT competition

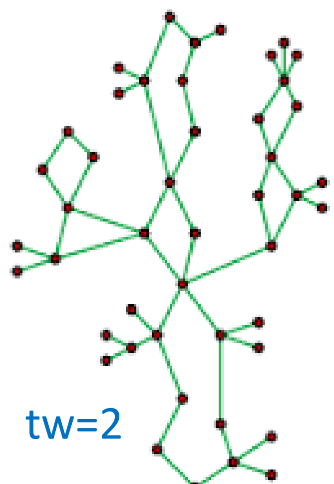
Solving SAT – Treewidth

- Several graph representations of CNF formulas exist
  - Representations capture variable-clause interactions

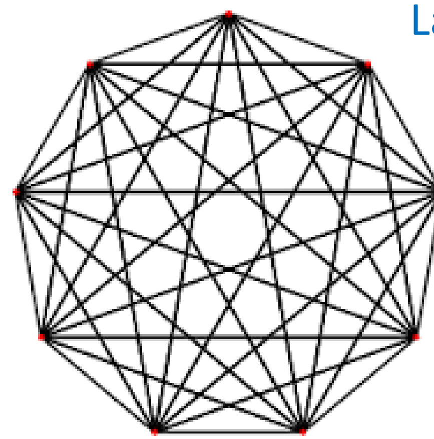
- SAT is FPT when parameterized by the **treewidth** of these graph representations
  - Standard dynamic programming



tw=1



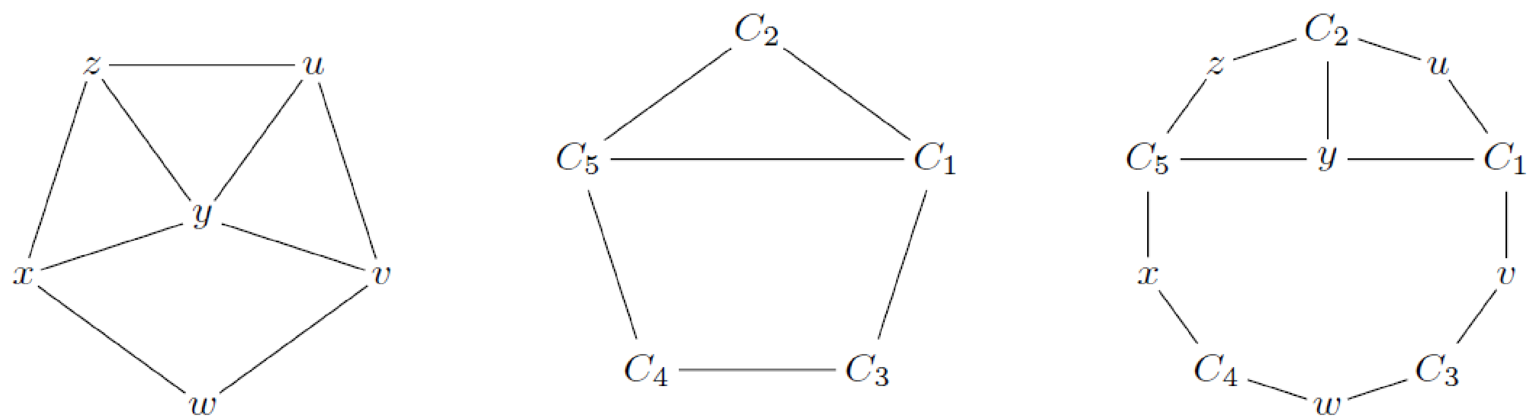
tw=2



Large tw

Graph Representations for SAT

- Example: $C_1 = (u \vee \neg v \vee y)$, $C_2 = (\neg u \vee z \vee \neg y)$, $C_3 = (v \vee \neg w)$, $C_4 = (w \vee \neg x)$, $C_5 = (x \vee y \vee \neg z)$
- Classical representations:

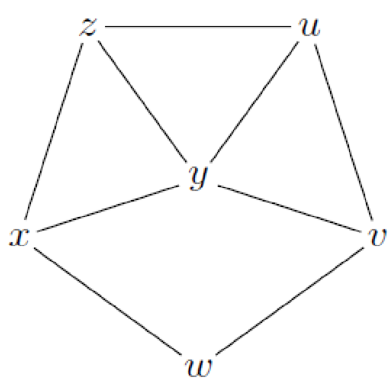


Primal graph   Dual graph   Incidence graph

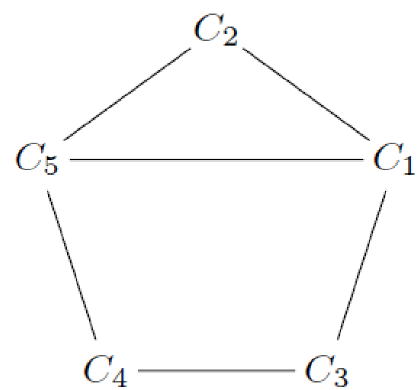
- Are there others?

Graph Representations for SAT

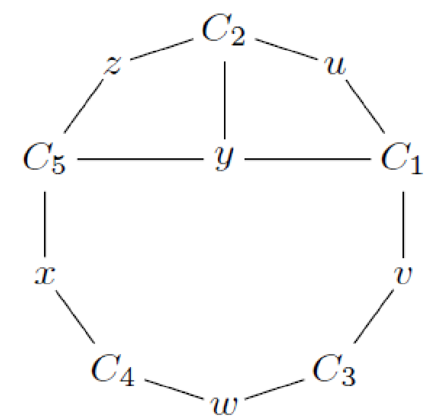
- Example: $C_1 = (u \vee \neg v \vee y)$, $C_2 = (\neg u \vee z \vee \neg y)$, $C_3 = (v \vee \neg w)$, $C_4 = (w \vee \neg x)$, $C_5 = (x \vee y \vee \neg z)$
- Classical representations:



Primal graph



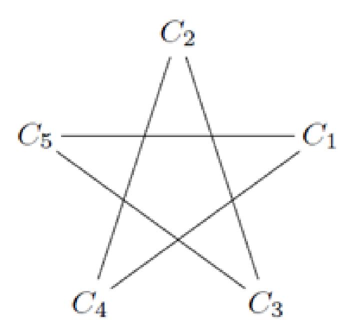
Dual graph



Incidence graph

- New representation:
  - Ganian, Szeider 2017

Edge ⟺ no contradicting literals



Consensus graph

Solving SAT – Treewidth

SAT is FPT parameterized by the treewidth of the primal/dual/incidence/consensus graph.

- Single-exponential runtime
- Better to use incidence graph rather than primal or dual
  - Can have much lower treewidth, opposite doesn't hold
- Good dynamic programming exercise
  - Consensus graph case is a bit more complicated

Solving SAT without Treewidth

- Tractable classes for SAT were studied for decades
  - Some are older than treewidth

- General idea: impose syntactic restrictions on clauses
  - Incomparable to the restrictions on variable-clause interactions imposed by treewidth

- Here, we focus on the two most prominent polynomial-time tractable classes for SAT:
  - **Horn**
  - **2CNF** (**Krom**)

Horn formulas

- **Each clause contains at most 1 positive literal**
- Example: $C_1 = (\neg z \vee \neg y)$, $C_2 = (u \vee \neg v \vee \neg y)$, $C_3 = (\neg u \vee z \vee \neg y)$, $C_4 = (b)$, $C_5 = (v \vee \neg b)$,

- Solving:
  1. Unit propagation
     - Unit clauses force a certain assignment $\longrightarrow$ apply it

Horn formulas

- **Each clause contains at most 1 positive literal**
- Example: $C_1 = (\neg z \vee \neg y)$, $C_2 = (u \vee \neg v \vee \neg y)$, $C_3 = (\neg u \vee z \vee \neg y)$, $C_4 = (1)$, $C_5 = (v \vee \neg 1)$,

- Solving:
  1. Unit propagation
     - Unit clauses force a certain assignment $\longrightarrow$ apply it

Horn formulas

- **Each clause contains at most 1 positive literal**
- Example: $C_1 = (\neg z \vee \neg y)$, $C_2 = (u \vee \neg v \vee \neg y)$, $C_3 = (\neg u \vee z \vee \neg y)$, $C_4 = (1)$, $C_5 = (v)$,

- Solving:
  1. Unit propagation
     - Unit clauses force a certain assignment $\longrightarrow$ apply it

Horn formulas

- **Each clause contains at most 1 positive literal**
- Example: $C_1 = (\neg z \vee \neg y)$, $C_2 = (u \vee \neg 1 \vee \neg y)$, $C_3 = (\neg u \vee z \vee \neg y)$, $C_4 = (1)$, $C_5 = (1)$,

- Solving:
  1. Unit propagation
     - Unit clauses force a certain assignment $\longrightarrow$ apply it

Horn formulas

- **Each clause contains at most 1 positive literal**
- Example: $C_1 = (\neg z \vee \neg y)$, $C_2 = (u \vee \neg y)$, $C_3 = (\neg u \vee z \vee \neg y)$, $C_4 = (1)$, $C_5 = (1)$,

- Solving:
  1. Unit propagation
     - Unit clauses force a certain assignment $\longrightarrow$ apply it

Horn formulas

- **Each clause contains at most 1 positive literal**
- Example: $C_1 = (\neg z \vee \neg y)$, $C_2 = (u \vee \neg y)$, $C_3 = (\neg u \vee z \vee \neg y)$, $C_4 = (1)$, $C_5 = (1)$,
- Solving:
  1. Unit propagation
     - Unit clauses force a certain assignment $\longrightarrow$ apply it
     - Afterwards, no unit clauses are left
  2. Assign all remaining variables to 0

Horn formulas

- **Each clause contains at most 1 positive literal**
- Example: $C_1 = (\neg 0 \lor \neg 0)$, $C_2 = (0 \lor \neg 0)$, $C_3 = (\neg 0 \lor 0 \lor \neg 0)$, $C_4 = (1)$, $C_5 = (1)$,
- Solving:
  1. Unit propagation
     - Unit clauses force a certain assignment $\longrightarrow$ apply it
     - Afterwards, no unit clauses are left
  2. Assign all remaining variables to 0

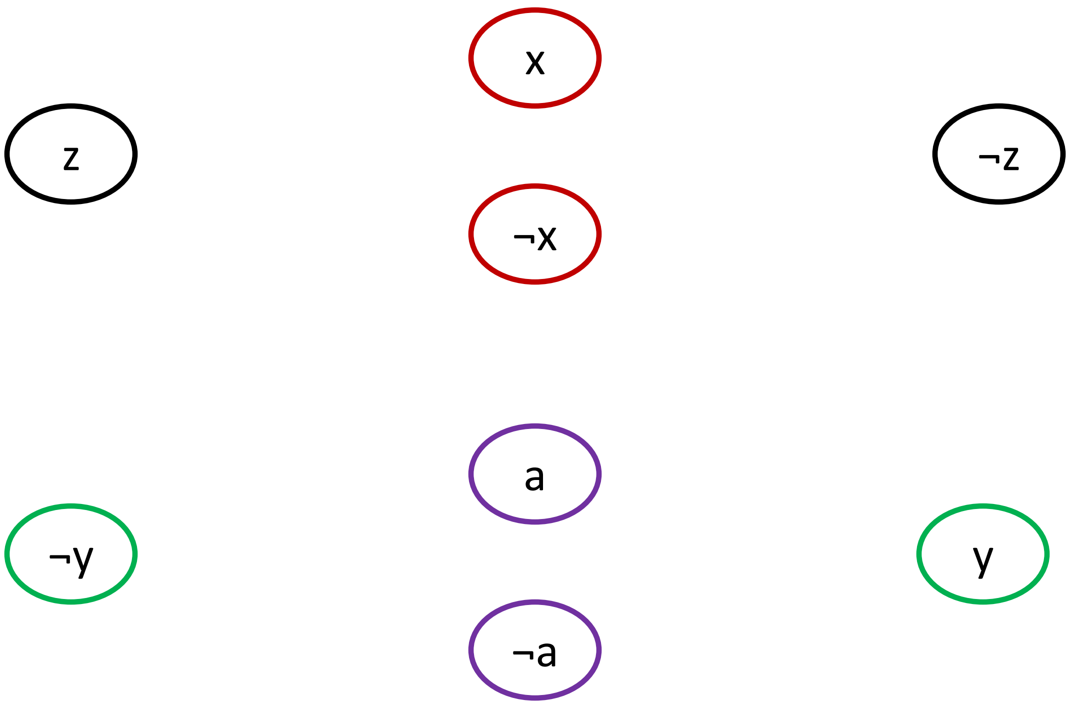
2CNF formulas

- **Each clause contains at most 2 literals**
- Example: $(\neg z \vee x) \wedge (y \vee a) \wedge (\neg z \vee \neg y) \wedge (z \vee y) \wedge (y \vee \neg a) \wedge (\neg z \vee \neg x)$

- For solving, we‘ll need the **implication graph**
  - 2 vertices per variable (positive / negative)
  - Edges represent implications arising from clauses

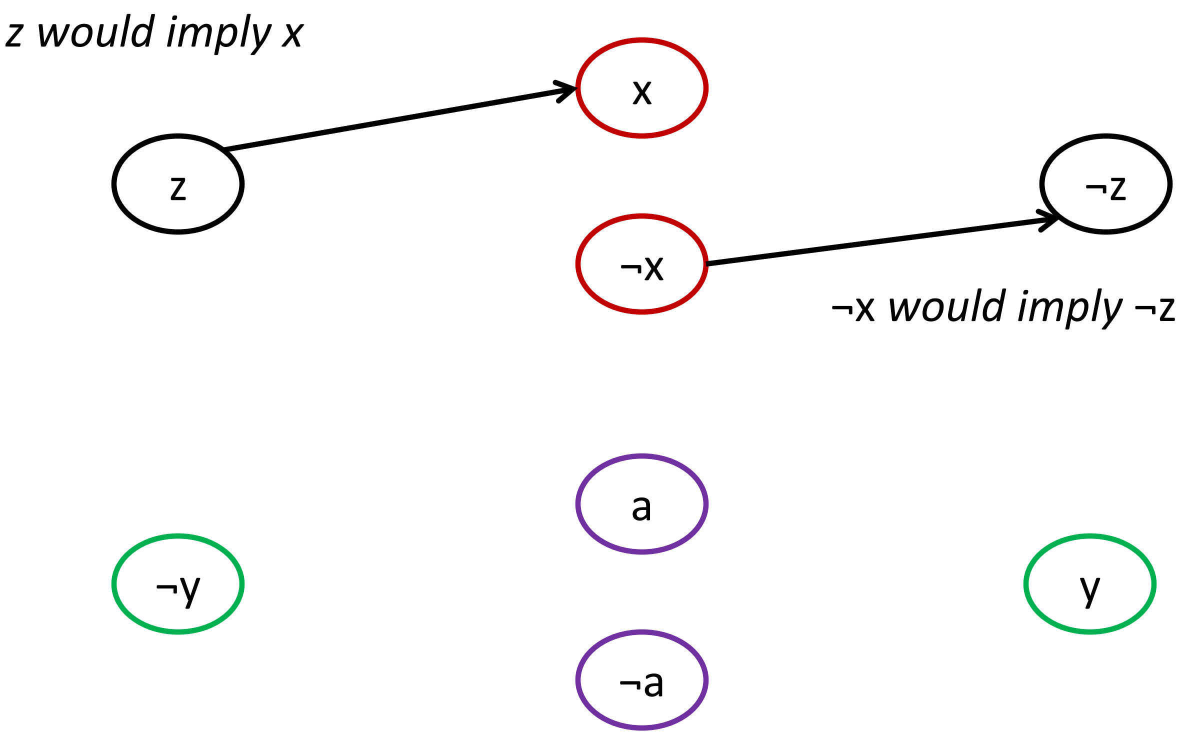
Implication Graph

$$(\neg z \lor x) \land (y \lor a) \land (\neg z \lor \neg y) \land (z \lor y) \land (y \lor \neg a) \land (\neg z \lor \neg x)$$



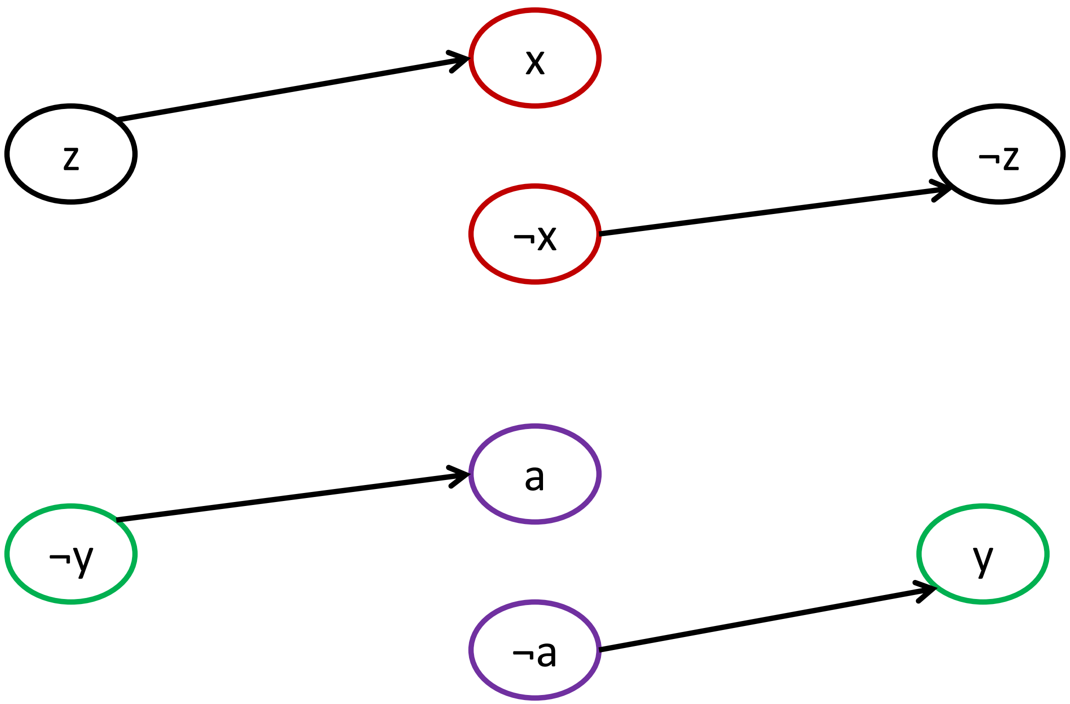
Implication Graph

$$(\neg z \vee x) \wedge (y \vee a) \wedge (\neg z \vee \neg y) \wedge (z \vee y) \wedge (y \vee \neg a) \wedge (\neg z \vee \neg x)$$



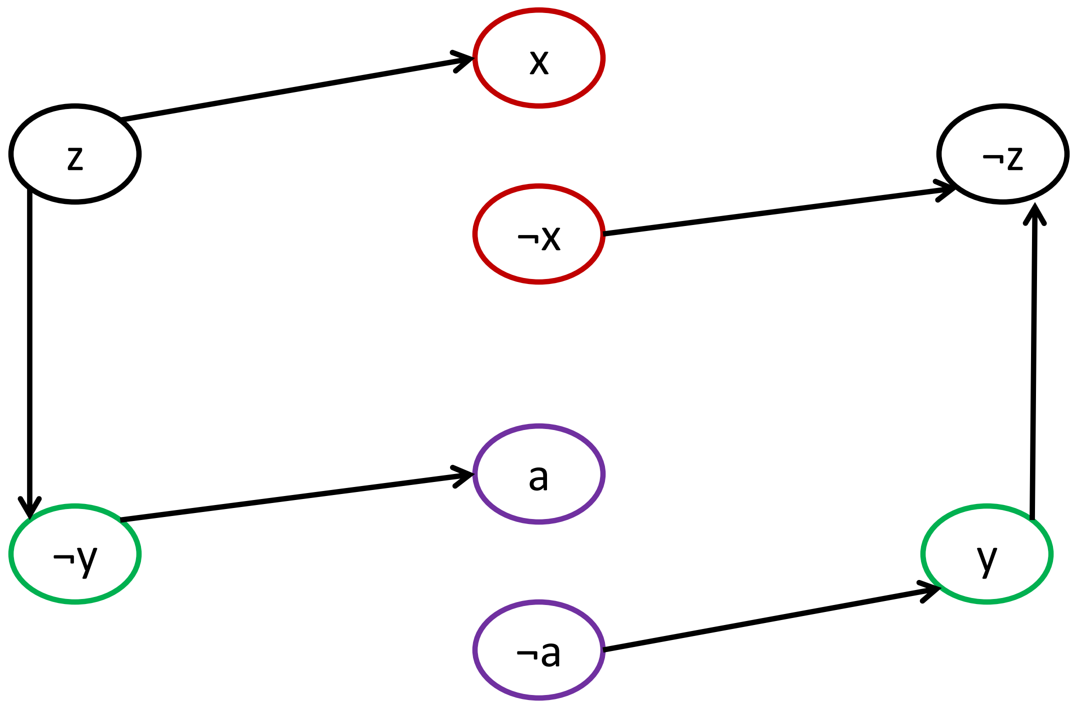
Implication Graph

$$(\neg z \vee x) \wedge (y \vee a) \wedge (\neg z \vee \neg y) \wedge (z \vee y)$$
$$\wedge (y \vee \neg a) \wedge (\neg z \vee \neg x)$$



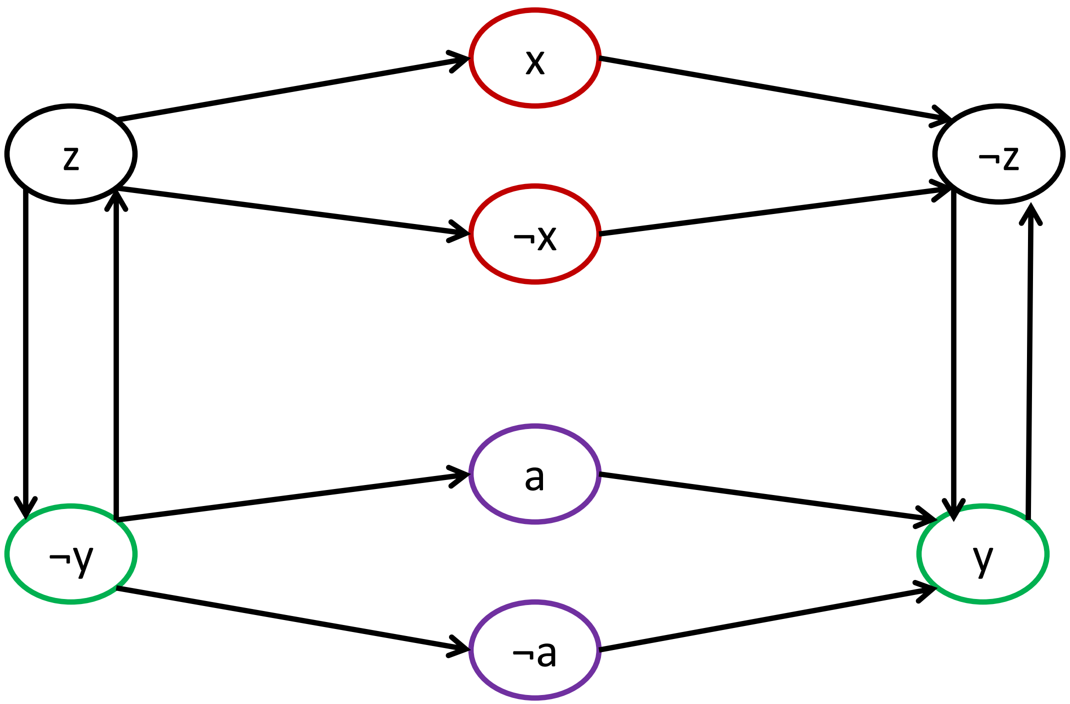
Implication Graph

$$(\neg z \lor x) \land (y \lor a) \land (\neg z \lor \neg y) \land (z \lor y) \land (y \lor \neg a) \land (\neg z \lor \neg x)$$



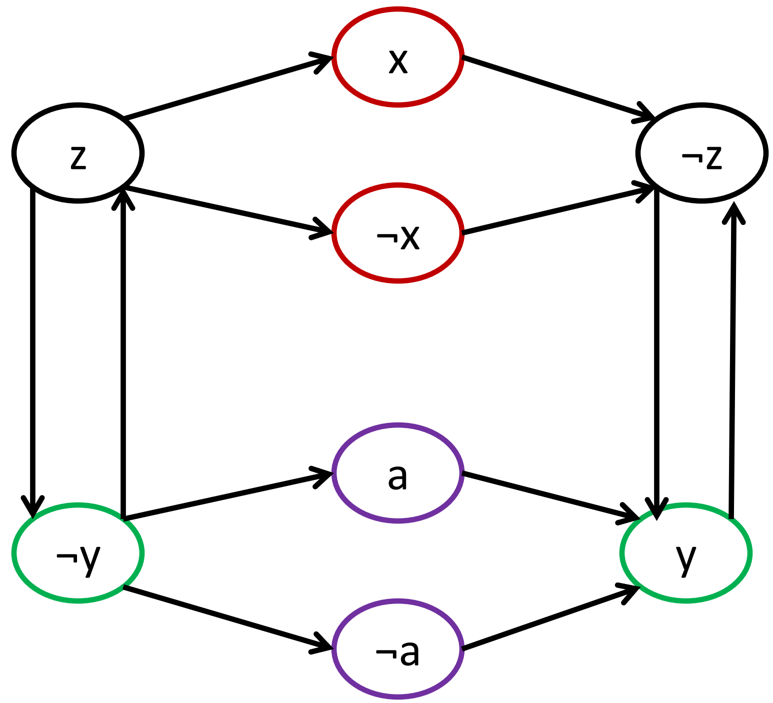
Implication Graph

$$(\neg z \lor x) \land (y \lor a) \land (\neg z \lor \neg y) \land (z \lor y) \land (y \lor \neg a) \land (\neg z \lor \neg x)$$



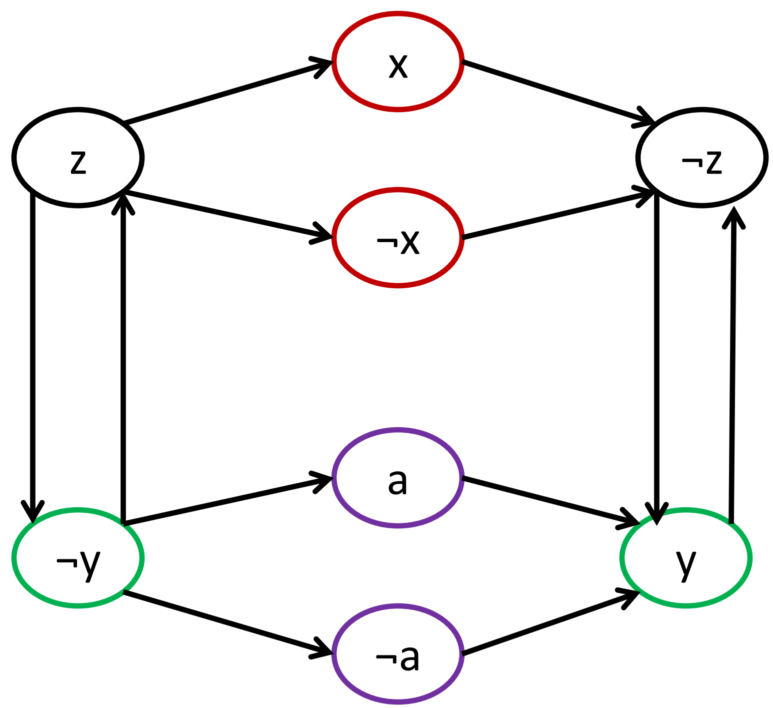
Solving 2CNF Formulas

- Example: $(\neg z \lor x) \land (y \lor a) \land (\neg z \lor \neg y) \land (z \lor y) \land (y \lor \neg a) \land (\neg z \lor \neg x)$
- Algorithm:
  1. Construct implication graph



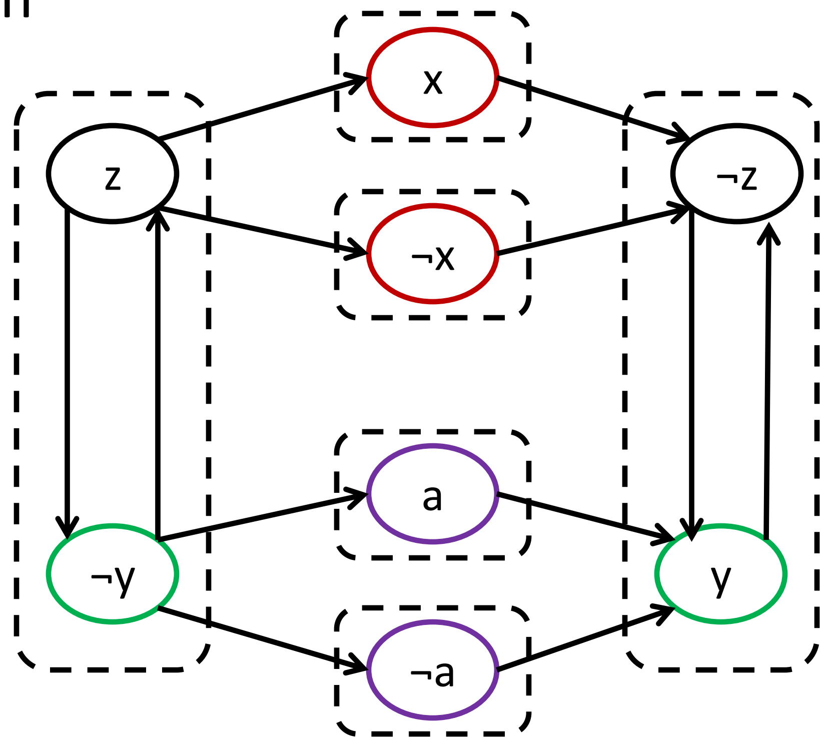
Solving 2CNF Formulas

- Example: $(\neg z \lor x) \land (y \lor a) \land (\neg z \lor \neg y) \land (z \lor y) \land (y \lor \neg a) \land (\neg z \lor \neg x)$
- Algorithm:
  1. Construct implication graph
  2. Find strongly connected components (SCCs)



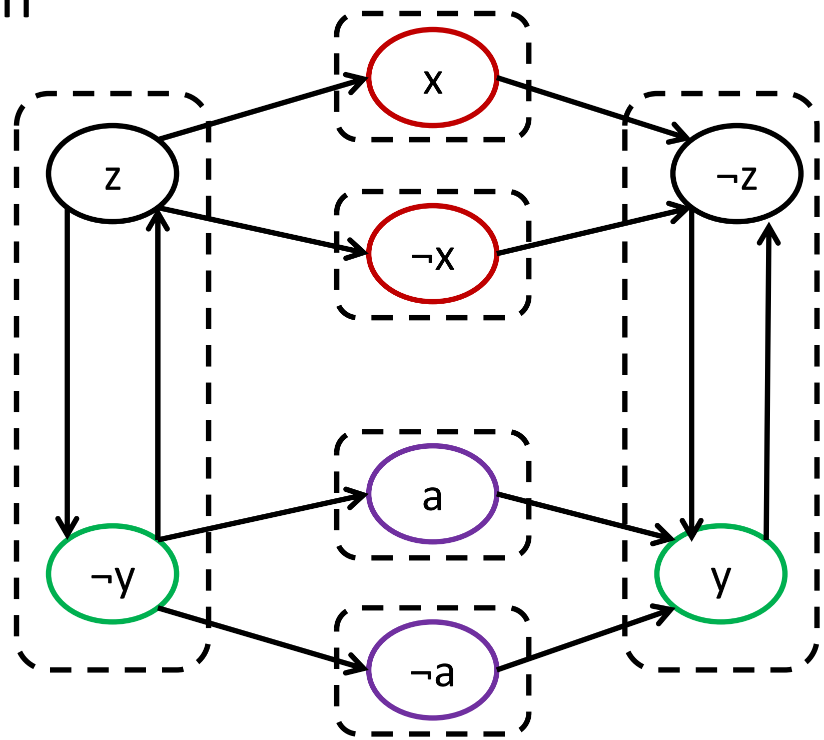
Solving 2CNF Formulas

- Example: $(\neg z \vee x) \wedge (y \vee a) \wedge (\neg z \vee \neg y) \wedge (z \vee y) \wedge (y \vee \neg a) \wedge (\neg z \vee \neg x)$
- Algorithm:
  1. Construct implication graph
  2. Find strongly connected components (SCCs)
     - If any SCC contains both literals for a variable, reject



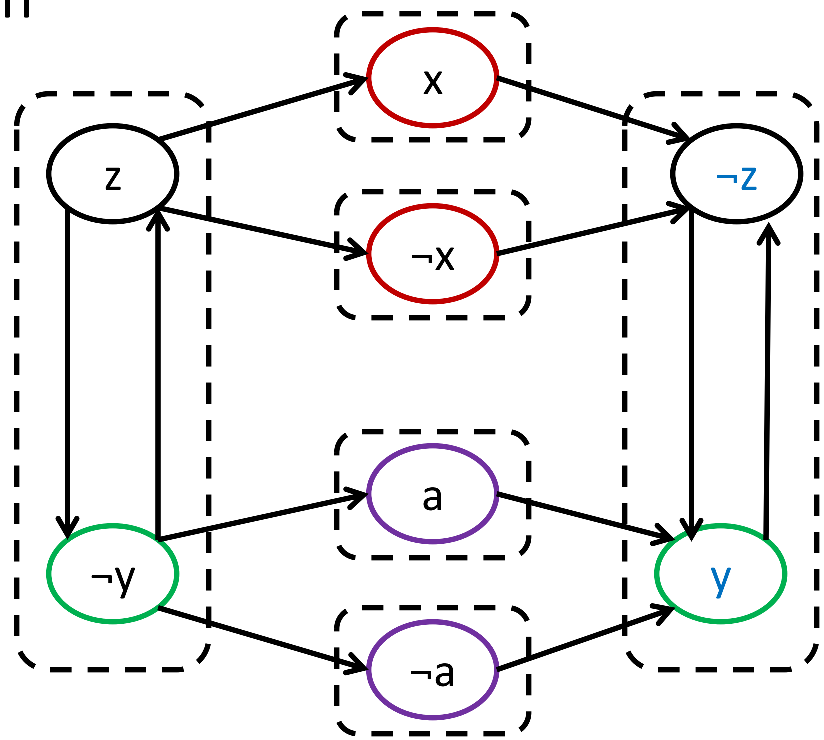
Solving 2CNF Formulas

- Example: $(\neg z \vee x) \wedge (y \vee a) \wedge (\neg z \vee \neg y) \wedge (z \vee y) \wedge (y \vee \neg a) \wedge (\neg z \vee \neg x)$
- Algorithm:
  1. Construct implication graph
  2. Find strongly connected components (SCCs)
     - If any SCC contains both literals for a variable, reject
  3. Start assigning literals to 1 from SCCs which are *sinks*



Solving 2CNF Formulas

- Example: $(1 \vee x) \wedge (1 \vee a) \wedge (1 \vee 0) \wedge (0 \vee 1) \wedge (1 \vee \neg a) \wedge (1 \vee \neg x)$
- Algorithm:
  1. Construct implication graph
  2. Find strongly connected components (SCCs)
     - If any SCC contains both literals for a variable, reject
  3. Start assigning literals to 1 from SCCs which are *sinks*
     - Continue until all clauses satisfied



Recap

SAT is polynomial-time tractable on 2CNF and Horn formulas.

- Result not covered by treewidth
  - Can easily construct an incidence graph that is a grid
- More general polynomial-time tractable classes exist
  - q-Horn, Renamable Horn, Hidden Extended Horn...
- But what does this have to do with PC and backdoors?
  - Backdoors allow us to measure *distance to triviality*
  - *Triviality* here means one of our **tractable classes** for SAT

Also called islands of tractability

Backdoor Motivation

- Consider the following formula *F*:

$$(\neg z \lor x \lor y) \land (x \lor \neg a) \land (\neg z \lor \neg x \lor \neg y)$$
$$\land (z \lor y \lor a) \land (\neg y \lor \neg a \lor x) \land (a \lor \neg x \lor y)$$

- **Claim**: *F* is *almost* a 2CNF formula
  - Just need to branch on assigning a single variable (y)
  - y $\longrightarrow$ 0:

$$(\neg z \lor x) \land (x \lor \neg a) \land (1) \land (z \lor a) \land (1) \land (a \lor \neg x)$$

  - y $\longrightarrow$ 1:

$$(1) \land (x \lor \neg a) \land (\neg z \lor \neg x) \land (1) \land (\neg a \lor x) \land (1)$$

Strong Backdoors

- A set **X** of variables is a **strong backdoor** to a tractable class **C** if each assignment of **X** results in a formula in **C**
- Parameter: size of a smallest strong backdoor to **C**
- General approach for fixed-parameter SAT solving:
  1. Find a size-k strong backdoor to a selected tractable class **C** (or identify that it doesn't exist)
  2. Use the strong backdoor to solve the instance
- Q: Why *strong*?

Weak Backdoors

- A set **X** of variables is a **weak backdoor** to a tractable class **C** if there exists an assignment of **X** which results in a *satisfiable* formula in **C**

Can be arbitrarily smaller than a strong backdoor

- Example: backdoors to 2CNF, many large clauses that can all be satisfied by setting a single variable to 0

Doesn’t exist for **NO**-instances

Detection usually **W[2]**-hard

In this talk we focus *mostly* on strong backdoors

Using Strong Backdoors

SAT can be solved in time $O^*(2^k)$ if a strong backdoor of size k to a tractable class C is provided on the input

- Simple branching over at most 2^k many assignments

Main difficulty: finding a strong backdoor to **C**

- Algorithms and techniques depend on **C**
- **XP** algorithm is trivial (assuming **C** is polynomial-time recognizable)

Backdoor Detection

- For Horn and 2CNF, we show equivalence to the simpler notion of *variable deletion*

X is a strong backdoor for Horn/2CNF iff deleting all occurrences of X results in a Horn/2CNF formula.

  - Sometimes called a **deletion backdoor**
  - For many classes, these are larger than strong backdoors

Example: $(\neg z \vee x \vee y) \wedge (x \vee \neg a) \wedge (\neg z \vee \neg x \vee \neg y)$
$\wedge (z \vee y \vee a) \wedge (\neg y \vee \neg a \vee x) \wedge (a \vee \neg x \vee y)$

Backdoor Detection

- For Horn and 2CNF, we show equivalence to the simpler notion of *variable deletion*

X is a strong backdoor for Horn/2CNF iff deleting all occurrences of X results in a Horn/2CNF formula.

  - Sometimes called a **deletion backdoor**
  - For many classes, these are larger than strong backdoors

Example: $(\neg z \vee x \vee y) \wedge (x \vee \neg a) \wedge (\neg z \vee \neg x \vee \neg y)$
$\wedge (z \vee y \vee a) \wedge (\neg y \vee \neg a \vee x) \wedge (a \vee \neg x \vee y)$

  - Let's try deleting *x*

Backdoor Detection

- For Horn and 2CNF, we show equivalence to the simpler notion of *variable deletion*

X is a strong backdoor for Horn/2CNF iff deleting all occurrences of X results in a Horn/2CNF formula.

  - Sometimes called a **deletion backdoor**
  - For many classes, these are larger than strong backdoors

Example: $(\neg z \vee y) \wedge (x \vee \neg a) \wedge (\neg z \vee \neg x \vee \neg y)$
$\wedge (z \vee y \vee a) \wedge (\neg y \vee \neg a \vee x) \wedge (a \vee \neg x \vee y)$

  - Let's try deleting *x*

Backdoor Detection

- For Horn and 2CNF, we show equivalence to the simpler notion of *variable deletion*

X is a strong backdoor for Horn/2CNF iff deleting all occurrences of X results in a Horn/2CNF formula.

- – Sometimes called a **deletion backdoor**
- – For many classes, these are larger than strong backdoors

Example: $(\neg z \vee y) \wedge (\neg a) \wedge (\neg z \vee \neg x \vee \neg y)$

$$\wedge (z \vee y \vee a) \wedge (\neg y \vee \neg a \vee x) \wedge (a \vee \neg x \vee y)$$

- – Let's try deleting *x*

Backdoor Detection

- For Horn and 2CNF, we show equivalence to the simpler notion of *variable deletion*

X is a strong backdoor for Horn/2CNF iff deleting all occurrences of X results in a Horn/2CNF formula.

  – Sometimes called a **deletion backdoor**
  – For many classes, these are larger than strong backdoors

Example: $(\neg z \vee y) \wedge (\neg a) \wedge (\neg z \vee \neg y)$

$$\wedge (z \vee y \vee a) \wedge (\neg y \vee \neg a \vee x) \wedge (a \vee \neg x \vee y)$$

  – Let's try deleting *x*

Backdoor Detection

- For Horn and 2CNF, we show equivalence to the simpler notion of *variable deletion*

X is a strong backdoor for Horn/2CNF iff deleting all occurrences of X results in a Horn/2CNF formula.

- Sometimes called a **deletion backdoor**
- For many classes, these are larger than strong backdoors

Example: $(\neg z \vee y) \wedge (\neg a) \wedge (\neg z \vee \neg y)$

$$\wedge (z \vee y \vee a) \wedge (\neg y \vee \neg a) \wedge (a \vee \neg x \vee y)$$

- Let's try deleting *x*

Backdoor Detection

- For Horn and 2CNF, we show equivalence to the simpler notion of *variable deletion*

X is a strong backdoor for Horn/2CNF iff deleting all occurrences of X results in a Horn/2CNF formula.

  - Sometimes called a **deletion backdoor**
  - For many classes, these are larger than strong backdoors

Example: $(\neg z \vee y) \wedge (\neg a) \wedge (\neg z \vee \neg y)$
$\wedge (z \vee y \vee a) \wedge (\neg y \vee \neg a) \wedge (a \vee y)$

  - Let's try deleting *x*

Deletion = Strong Backdoors

X is a strong backdoor for Horn/2CNF iff deleting all occurrences of X results in a Horn/2CNF formula.

- **X** is strong: For each clause *d*, there is an assignment to X which doesn't satisfy *d*, hence *d*-**X** must be Horn/2CNF

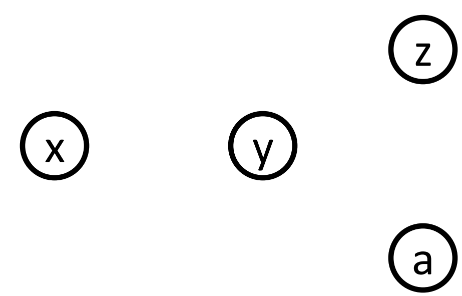
- **X** is a deletion set: For each clause *d*, we know that *d*-**X** is Horn/2CNF. Each assignment to **X** will either delete *d* or result in *d*-**X** for this clause.

Backdoor Detection: Horn

- We reduce the deletion problem to **Vertex Cover**

Example: $(\neg z \vee x \vee y) \wedge (x \vee \neg a) \wedge (\neg z \vee \neg x \vee \neg y)$
$\wedge (z \vee y \vee a) \wedge (\neg y \vee \neg a \vee x) \wedge (a \vee \neg x \vee y)$

- Construct a graph **G** as follows:
  - Variables are vertices...



Backdoor Detection: Horn

- We reduce the deletion problem to **Vertex Cover**

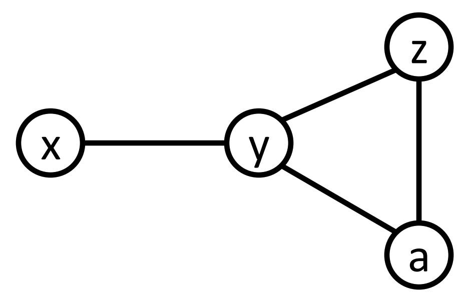
Example: $(\neg z \vee x \vee y) \wedge (x \vee \neg a) \wedge (\neg z \vee \neg x \vee \neg y) \wedge (z \vee y \vee a) \wedge (\neg y \vee \neg a \vee x) \wedge (a \vee \neg x \vee y)$

- Construct a graph **G** as follows:
  - Variables are vertices...
  - Add edge if both variables occur positively in some clause

Vertex cover in **G**

$\Updownarrow$

Deletion backdoor to Horn



Backdoor Detection: 2CNF

- We reduce the deletion problem to **3-Hitting Set**
  - Note: could also use bounded search trees

Example: $(\neg a \vee e \vee c) \wedge (d \vee e) \wedge (\neg b \vee \neg c \vee \neg d)$
$\wedge (d \vee c \vee \neg a \vee b) \wedge (b \vee \neg e \vee a)$

- Construct a **3-Hitting Set** instance **H** as follows:
  - Ground set is the set of variables
  - Target sets are all triples which occur together in a clause
  - For our example: *{ace}, {abc}, {abd}, {acd}, {bcd}, {abe}*

Hitting Set $\Longleftrightarrow$ Deletion backdoor to Horn

Strong Backdoors: Summary

SAT can be solved in time $O^*(2^k)$ parameterized by the size of a strong backdoor to Horn.

– Runtime: $O^*(1.3^k)$ for finding and then $O^*(2^k)$ for using
  - Uses **Vertex Cover** algorithm of Chen, Kanj and Xia [2010]

SAT can be solved in time $O^*(2.27^k)$ parameterized by the size of a strong backdoor to 2CNF.

– Runtime: $O^*(2.27^k)$ for finding and then $O^*(2^k)$ for using
  - Uses **3-Hitting Set** algorithm of Niedermeier, Rossmanith [2003]

Intermezzo: Weak BD Detection

- Why is weak backdoor detection harder?

- **Recall**:

A set **X** of variables is a **weak backdoor** to a tractable class **C** if there exists an assignment of **X** which results in a *satisfiable* formula in **C**

Intermezzo: Weak BD Detection

- Why is weak backdoor detection harder?
- Intuition: weak backdoors can "kill" large obstructions with a single variable

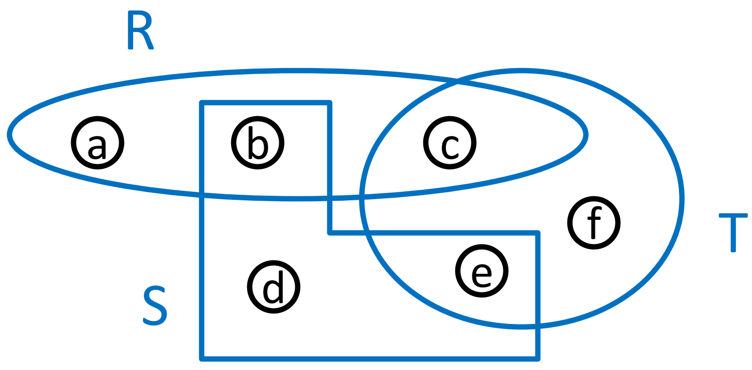
↓

- Can't reliably find small obstructions to branch on
- Example: Weak BD detection to Horn is W[2]-hard
- Proof: Reduction from **Hitting Set**

General template

The Reduction

- Starting point: Hitting Set instance **S**, parameter **k**

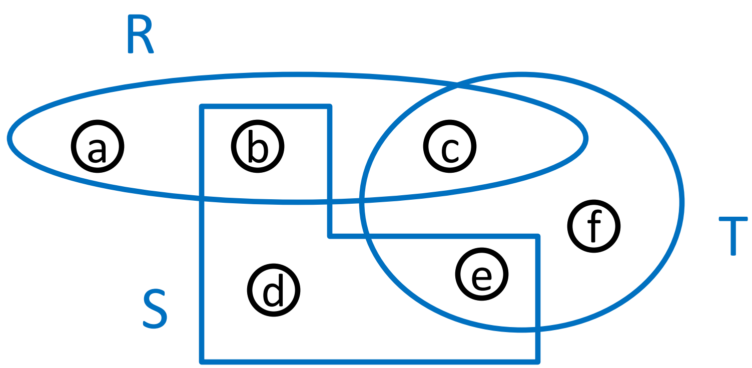


k=2

- Elements -> *main* variables
- For each set (R,S,T), we create k+1 clauses such that:
  - they are not Horn
  - they can be satisfied by any element (variable) in the set
  - they contain auxiliary variables which shouldn't be in a BD

The Reduction

- Starting point: Hitting Set instance **S**, parameter **k**



k=2

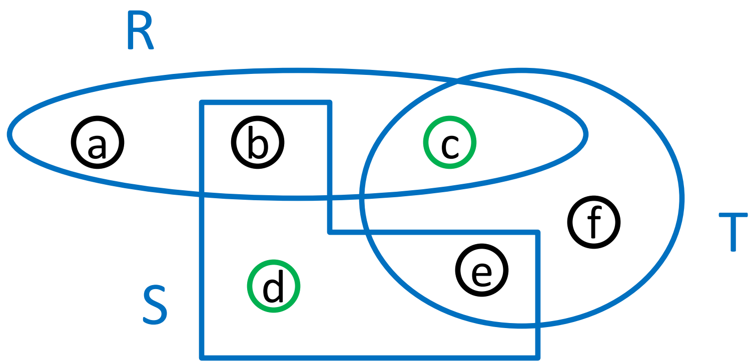
- Clauses:

$$(r_1 \vee a \vee b \vee c) \wedge (r_2 \vee a \vee b \vee c) \wedge (r_3 \vee a \vee b \vee c)$$
$$\wedge (s_1 \vee b \vee d \vee e) \wedge (s_2 \vee b \vee d \vee e) \wedge (s_3 \vee b \vee d \vee e)$$
$$\wedge (t_1 \vee c \vee e \vee f) \wedge (t_2 \vee c \vee e \vee f) \wedge (t_3 \vee c \vee e \vee f)$$

– Taking any variables other than a,b,c,d,e,f is suboptimal

The Reduction

- Starting point: Hitting Set instance **S**, parameter **k**



k=2

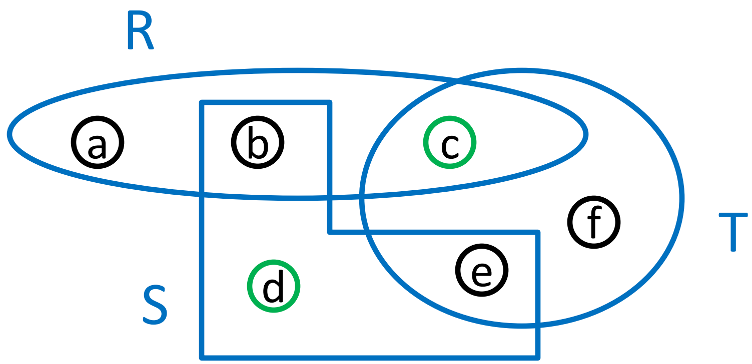
Consider a **Hitting Set** solution

- Clauses:

$$(r_1 \vee a \vee b \vee c) \wedge (r_2 \vee a \vee b \vee c) \wedge (r_3 \vee a \vee b \vee c)$$
$$\wedge (s_1 \vee b \vee d \vee e) \wedge (s_2 \vee b \vee d \vee e) \wedge (s_3 \vee b \vee d \vee e)$$
$$\wedge (t_1 \vee c \vee e \vee f) \wedge (t_2 \vee c \vee e \vee f) \wedge (t_3 \vee c \vee e \vee f)$$

The Reduction

- Starting point: Hitting Set instance **S**, parameter **k**



k=2

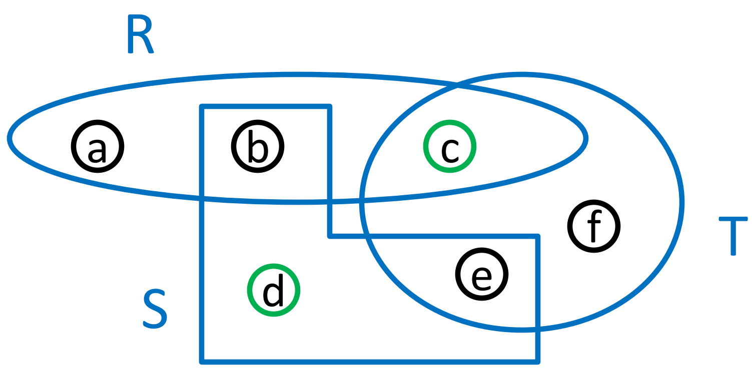
Consider a **Hitting Set** solution

- Clauses:

$$(r_1 \vee a \vee b \vee c) \wedge (r_2 \vee a \vee b \vee c) \wedge (r_3 \vee a \vee b \vee c)$$
$$\wedge (s_1 \vee b \vee d \vee e) \wedge (s_2 \vee b \vee d \vee e) \wedge (s_3 \vee b \vee d \vee e)$$
$$\wedge (t_1 \vee c \vee e \vee f) \wedge (t_2 \vee c \vee e \vee f) \wedge (t_3 \vee c \vee e \vee f)$$

The Reduction

- Starting point: Hitting Set instance **S**, parameter **k**



k=2

Consider a **Hitting Set** solution

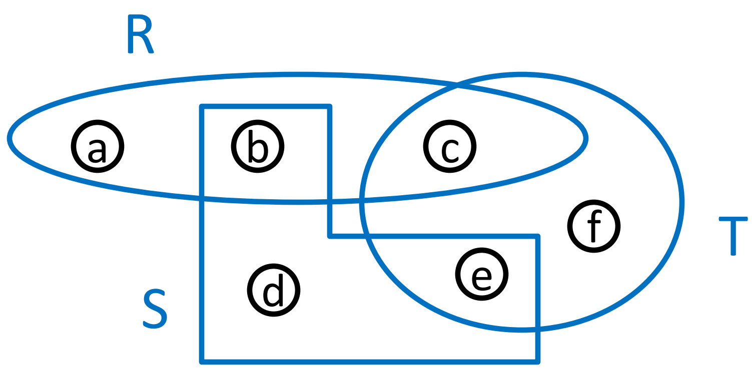
- Clauses:

$$(r_1 \vee a \vee b \vee 1) \wedge (r_2 \vee a \vee b \vee 1) \wedge (r_3 \vee a \vee b \vee 1)$$
$$\wedge (s_1 \vee b \vee 1 \vee e) \wedge (s_2 \vee b \vee 1 \vee e) \wedge (s_3 \vee b \vee 1 \vee e)$$
$$\wedge (t_1 \vee 1 \vee e \vee f) \wedge (t_2 \vee 1 \vee e \vee f) \wedge (t_3 \vee 1 \vee e \vee f)$$

  – We obtain a weak backdoor of size at most **k**

The Reduction

- Starting point: Hitting Set instance **S**, parameter **k**



k=2

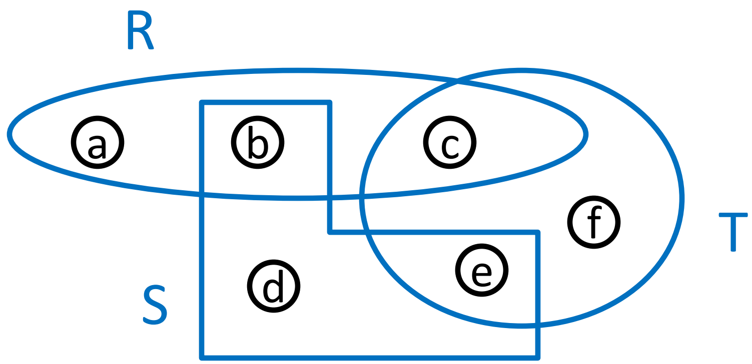
Consider a **Weak Backdoor X** of size $\leq$k

- Clauses:

$$(r_1 \vee a \vee b \vee c) \wedge (r_2 \vee a \vee b \vee c) \wedge (r_3 \vee a \vee b \vee c)$$
$$\wedge (s_1 \vee b \vee d \vee e) \wedge (s_2 \vee b \vee d \vee e) \wedge (s_3 \vee b \vee d \vee e)$$
$$\wedge (t_1 \vee c \vee e \vee f) \wedge (t_2 \vee c \vee e \vee f) \wedge (t_3 \vee c \vee e \vee f)$$

The Reduction

- Starting point: Hitting Set instance **S**, parameter **k**



k=2

Consider a **Weak Backdoor X** of size ≤k

- Clauses:

$$(r_1 \lor a \lor b \lor c) \land (r_2 \lor a \lor b \lor c) \land (r_3 \lor a \lor b \lor c)$$
$$\land (s_1 \lor b \lor d \lor e) \land (s_2 \lor b \lor d \lor e) \land (s_3 \lor b \lor d \lor e)$$
$$\land (t_1 \lor c \lor e \lor f) \land (t_2 \lor c \lor e \lor f) \land (t_3 \lor c \lor e \lor f)$$

  – Can assume **X** disjoint from red variables

  – **X** must intersect each of (R,S,T)

X is a Hitting Set

Better Backdoors

- Consider the following example:

$$\mathbf{F} = (\neg a \vee b \vee c) \wedge (\neg a \vee b \vee d) \wedge (\neg a \vee c \vee e) \wedge (\neg a \vee d \vee e) \wedge (a \vee \neg b \vee c \vee \neg d \vee \neg e) \wedge (a \vee b \vee \neg c \vee \neg e) \wedge (a \vee \neg b \vee \neg c \vee \neg d \vee e) \wedge (a \vee \neg b \vee \neg c \vee d)$$

- **F** has no small strong backdoor to Horn or 2CNF
- But what happens if we try assigning a?

Better Backdoors

a = 1

$\downarrow$

F = $(b \vee c) \wedge (b \vee d) \wedge (c \vee e) \wedge (d \vee e)$

a = 0

$\downarrow$

F = $(\neg b \vee c \vee \neg d \vee \neg e) \wedge (b \vee \neg c \vee \neg e) \wedge$
$(\neg b \vee \neg c \vee \neg d \vee e) \wedge (\neg b \vee \neg c \vee d)$

Better Backdoors

a = 1

$$\mathbf{F} = (b \lor c) \land (b \lor d) \land (c \lor e) \land (d \lor e)$$

2CNF

a = 0

Horn

$$\mathbf{F} = (\neg b \lor c \lor \neg d \lor \neg e) \land (b \lor \neg c \lor \neg e) \land (\neg b \lor \neg c \lor \neg d \lor e) \land (\neg b \lor \neg c \lor d)$$

Heterogeneous Backdoors

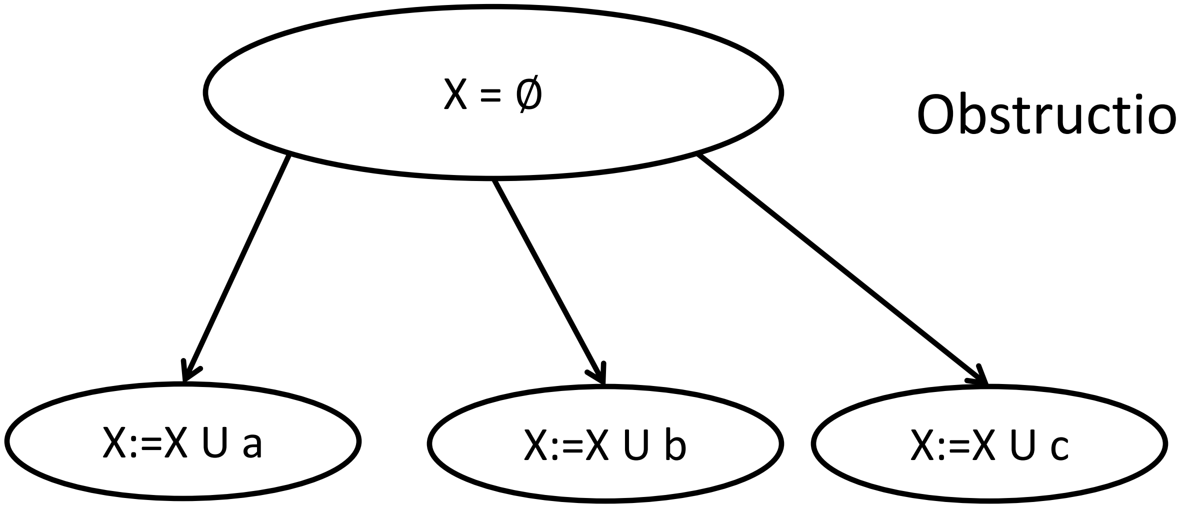
- A set **X** of variables is a **heterogeneous backdoor** to tractable classes $\mathbf{\{C_1, C_2, \ldots\}}$ if each assignment of **X** results in a formula in some $\mathbf{C_i}$
  - Gaspers, Misra, Ordyniak, Szeider, Zivny (2014)

- As easy to use as standard strong backdoors

- What about detection (finding)?

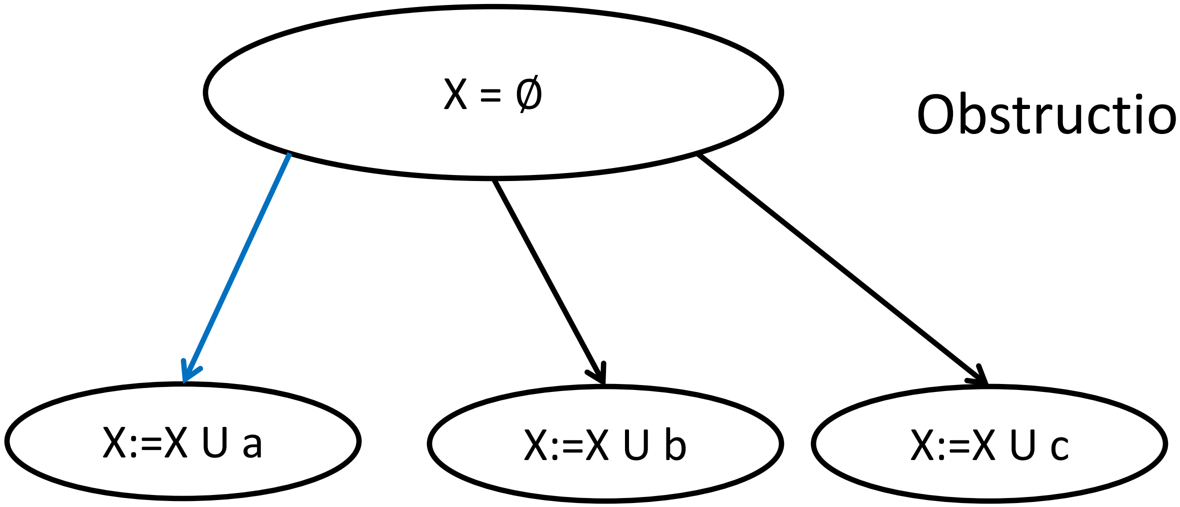
Finding Heterogeneous Backdoors

- Let's set **C = {2CNF,Horn}**
  – This means we'll be searching for a set of variables X such that each assignment to X results in a 2CNF or Horn formula
  – Main idea: Find an obstruction and branch on how to fix it



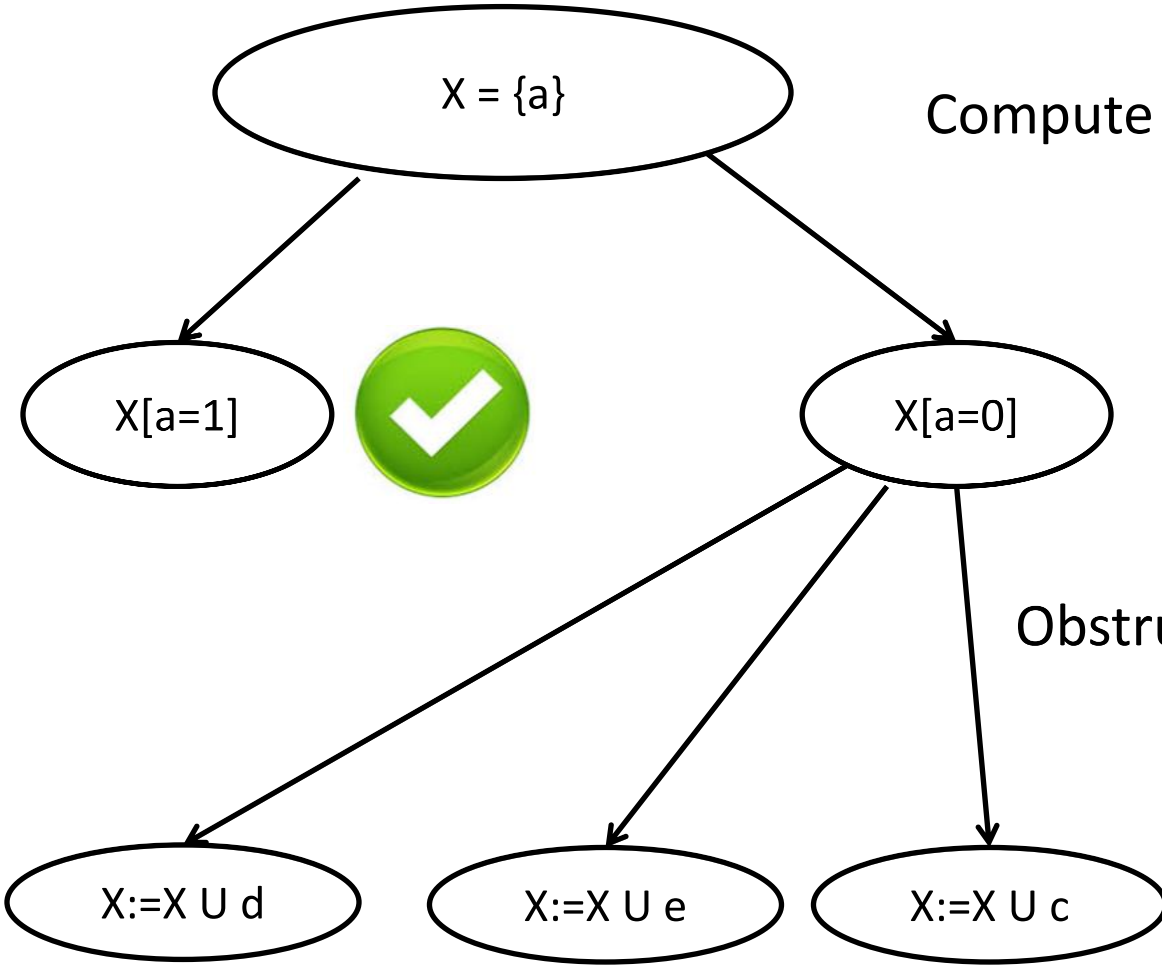
Obstruction over variables a,b,c

Branch



Obstruction over variables a,b,c

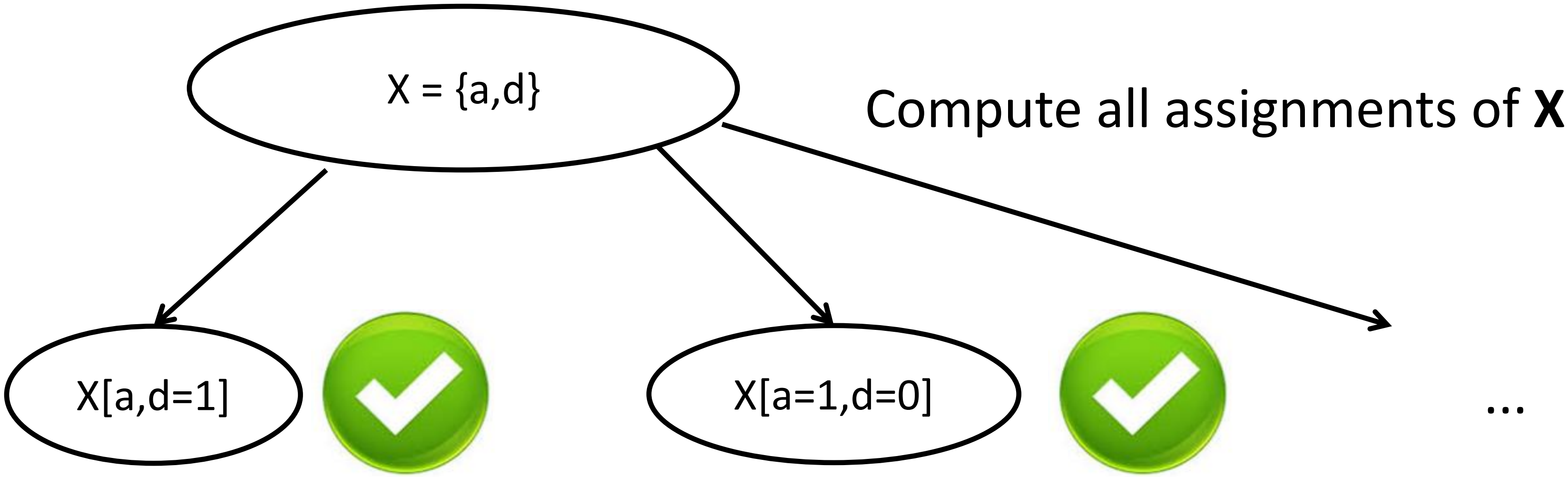
Branch



Compute all assignments of **X**

Obstruction over variables d,e,c

Branch


X = {a,d}
Compute all assignments of **X**
X[a,d=1]
X[a=1,d=0]
...

Obstructions for {2CNF,Horn}

- **Case 1**: clause that is neither 2CNF nor Horn
  - Example: $(z \vee y \vee a \vee \neg b)$
  - Must contain at least 2 positive literals and have size at least 3
  - Obstruction: an arbitrary set of 3 variables occurring in the clause, 2 of which occur positively

Obstructions for {2CNF,Horn}

- **Case 1**: clause that is neither 2CNF nor Horn
  - Example: $(z \vee y \vee a \vee \neg b)$
  - Must contain at least 2 positive literals and have size at least 3
  - Obstruction: an arbitrary set of 3 variables occurring in the clause, 2 of which occur positively (here: *y*, *a*, *b*)
  - Branching factor: 3

Obstructions for {2CNF,Horn}

- **Case 2**: the formula is neither "fully" Horn nor 2CNF
  - Choose 1 clause that's only Horn and one that's only 2CNF
  - Example: C_1= $(z \vee \neg y \vee \neg a \vee \neg b)$, C_2= $(y \vee x)$
  - **X** must either transform C_1 to 2CNF or C_2 to Horn
    - C_2 contains at most 2 literals
    - C_1 can be large, but any 3 literals form an obstruction to 2CNF
  - Branching factor: at most 5
    - here: z, y, a, x

Obstructions for {2CNF,Horn}

- **Case 3**: the formula is either “fully” Horn or 2CNF
  - Means this branch is ok

- **Runtime bound**:

$$5 \cdot (2^1 n + 5 \cdot (2^2 n + 5 \cdot (2^3 n + \cdots))) =$$
$$5^{O(k)} n = 2^{O(k)} n$$

Complexity map for other islands of tractability is known (FPT / W-hard).

Constraint Satisfaction (CSP)

- Introduced by Montanari in 1974
- Focus of intensive research (AI, TCS, Combinatorics, Algebra…)
- Dedicated conference

Problem Definition

- Instance: **I=(V,D,C)** where
  - **V** is a set of **variables**
  - **D** is a set of values (the **domain**)
  - **C** is a set of **constraints**

- Each constraint consists of a *scope* **S** and *relation* **R**
  - **S** is a tuple of variables (that the constraint applies to)
  - **R** encodes admissible values of **S**

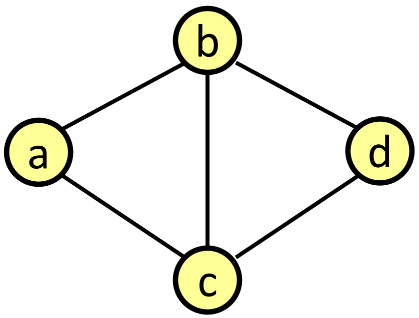
Constraint encoding **XOR(x,y)**

x	y
0	1
1	0

Problem Definition

- An **assignment** is a mapping **f**: **V** → **D**
- An assignment **satisfies** a CSP instance if for each constraint $(S=(x_1,\ldots x_r),R)$ we have $(f(x_1),\ldots,f(x_r)) \in R$.
- A CSP instance is **satisfiable** if it has at least one satisfying assignment
- The CSP problem asks whether the input instance is **satisfiable**
- CSP directly generalizes many known NP-complete problems

Example: 3-Coloring



Is it possible to color **a**,**b**,**c**,**d** by red, blue, green so that neighbors always get different colors?

V={**a**,**b**,**c**,**d**}
D={red,blue,green}

C={$\mathbf{c_{ab}}$, $\mathbf{c_{ac}}$, $\mathbf{c_{bc}}$, $\mathbf{c_{bd}}$, $\mathbf{c_{cd}}$}

Each $\mathbf{c_{xy}}$ contains the relation

x	y
red	blue
blue	red
blue	green
green	blue
red	green
green	red

CSP vs SAT

SAT

- Each clause **prevents** 1 assignment

$$(x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5 \vee x_6)$$

CSP

- Each tuple in a constraint **enables** 1 assignment

x_1	x_2	x_3	x_4	x_5	x_6
0	0	0	0	0	0
1	1	1	1	1	1

Solving CSP

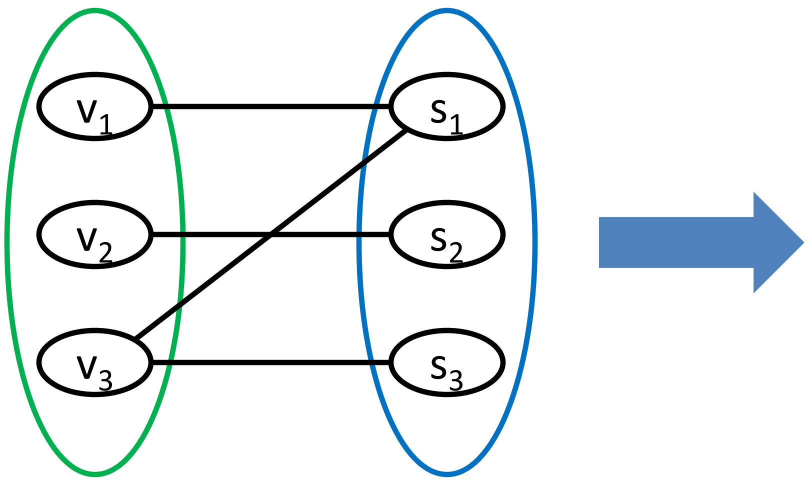
- Can define graph representations similarly as for SAT
  - Primal graphs, dual graphs, incidence graphs...
- Can also define backdoors (to some tractable classes)

But do these actually help us solve CSP?

- Two cases: bounded vs. unbounded domain
  - Constant-size vs. part of input

Unbounded Domain

- Can encode Multicolored Clique using **k** variables
  - One variable for each color
  - Constraints encode edges



Domain: {1,2,3}

Variables: g, b

g	b
1	1
2	2
3	1
3	3

Unbounded Domain

- Can encode Multicolored Clique using **k** variables
  - One variable for each color
  - Constraints encode edges between colors (at most $\mathbf{k^2}$)

⇒ W[1]-hard parameterized by treewidth

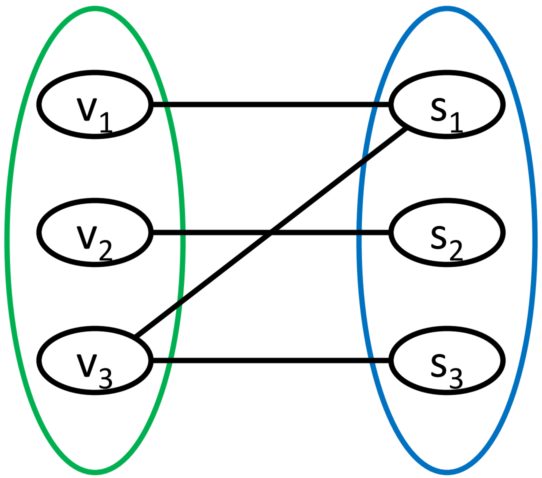
- Holds for primal, dual, incidence graph representations
- XP algorithm known

⇒ W[1]-hard parameterized by backdoors

- Holds regardless of selected island of tractability
- Brute-force XP algorithm

Bounded Domain

- Can encode MCC using **k^2** constraints
  - One binary variable for each vertex
  - Constraints ensure only one activated for each color
  - Constraints ensure we get a clique



Domain: {0,1}

Variables: v_1, v_2, v_3, s_1, s_2, s_3

v_1	v_2	v_3	s_1	s_2	s_3
1	0	0	1	0	0
0	1	0	0	1	0
0	0	1	1	0	0
0	0	1	0	0	1

Bounded Domain

- Can encode MCC using **k^2+k** constraints
  - One binary variable for each vertex
  - Constraints ensure only one activated for each color
  - Constraints ensure we get a clique

⇒ W[1]-hard par. by incidence and dual treewidth

  - FPT par. by primal treewidth (standard dyn. programming)

Bounded Domain

- If we are given a (strong) backdoor to any island **C**:
  - FPT algorithm – runtime: $|D|^k \cdot n^{O(1)}$
  - Holds for each island of tractability **C**
- But what are the islands of tractability for CSP?
  - Main direction: definition via **languages**
  - Language = set of relations that can be used in constraints
  - Example: Boolean language Γ:

0	0
1	1
1	0

0	0
1	1
0	1

0	1
1	0
1	1

0	1
1	0
0	0

  - CSP[Γ] is precisely 2CNF.

Schaefer's Theorem

For every finite Boolean language Γ: either Γ satisfies one of Schaefer's **polymorphisms** and CSP[Γ] is in **P**, or CSP[Γ] is **NP**-complete.

- Polymorphism: a procedure for constructing a new tuple from a fixed number of tuples in a relation
  - New tuple is built "column-by-column" by the same rule
- Γ satisfies a polymorphism δ iff Γ is closed under δ
- Example: **Majority** polymorphism
  - Take 3 tuples, rule for new columns: take what occurs most frequently in that column

0	0	1
1	1	1
1	0	0

$\Rightarrow$

1	0	1

Schaefer's Theorem: Exercise

For every finite Boolean language Γ: either Γ satisfies one of Schaefer's **polymorphisms** and CSP[Γ] is in **P**, or CSP[Γ] is **NP**-complete.

- Schaefer's Theorem implies tractability of 2CNF
  - Recall the ternary **Majority** polymorphism
  - Each 2CNF formula is equivalent to an instance of CSP[Γ]

0	0
1	1
1	0

0	0
1	1
0	1

0	1
1	0
1	1

0	1
1	0
0	0

  - Each of the above relations is closed under **Majority**

⇒ Γ satisfies **Majority** and CSP[Γ] is in **P**

Schaefer's Theorem: Islands

For every finite Boolean language Γ: either Γ satisfies one of Schaefer's **polymorphisms** and CSP[Γ] is in **P**, or CSP[Γ] is **NP**-complete.

- Schaefer's Theorem leads to 6 islands of tractability
  1. 0-valid
  2. 1-valid
  3. Horn
  4. Anti-Horn
  5. Affine
  6. Bijunctive (2CNF)

Beyond Schaefer

- **Feder-Vardi Conjecture**: extension of Schaefer’s Theorem to all finite languages
    - Remark: finite language $\Rightarrow$ bounded domain and arity

For every finite language Γ: either CSP[Γ] is in **P** or **NP**-complete.

– Recently settled (Bulatov; Zhuk 2017)

- **Bulatov’s Conservative Dichotomy**:

For every finite **conservative** language Γ: either Γ satisfies certain polymorphisms and CSP[Γ] is in **P**, or CSP[Γ] is **NP**-complete.

– Conservative = includes all unary relations

= allows domain restrictions

0
2

Languages and Backdoors

For every finite language Γ, strong backdoor detection to CSP[Γ] is FPT parameterized by backdoor size.

– **Recall**: variable set **X** is a strong backdoor if each assignment of X results in an instance of CSP[Γ]

– **Observation**: assume Γ has maximum arity of *c* and we're searching for a backdoor of size *k* in instance **I**

v_1	v_2	v_3	v_4	v_5	v_6
1	0	0	1	0	2
0	1	0	0	2	0
2	0	2	0	0	1
0	2	1	0	0	1

k = 3, c = 2

Constraint of arity > k+c

Languages and Backdoors

For every finite language Γ, strong backdoor detection to CSP[Γ] is FPT parameterized by backdoor size.

- **Recall**: variable set **X** is a strong backdoor if each assignment of X results in an instance of CSP[Γ]
- **Observation**: assume Γ has maximum arity of *c* and we're searching for a backdoor of size *k* in instance **I**

v_1	v_2	v_3	v_4	v_5	v_6
1	0	0	1	0	2
0	1	0	0	2	0
2	0	2	0	0	1
0	2	1	0	0	1

$v_4 = 0$
$v_5 = 0$
$v_6 = 0$

k = 3, c = 2

Constraint of arity > k+c

Languages and Backdoors

For every finite language Γ, strong backdoor detection to CSP[Γ] is FPT parameterized by backdoor size.

– **Recall**: variable set **X** is a strong backdoor if each assignment of X results in an instance of CSP[Γ]

– **Observation**: assume Γ has maximum arity of *c* and we're searching for a backdoor of size *k* in instance **I**

v_1	v_2	v_3

$v_4 = 0$
$v_5 = 0$
$v_6 = 0$

k = 3, c = 2

Languages and Backdoors

For every finite language Γ, strong backdoor detection to CSP[Γ] is FPT parameterized by backdoor size.

- **Recall**: variable set **X** is a strong backdoor if each assignment of X results in an instance of CSP[Γ]
- **Observation**: assume Γ has maximum arity of *c* and we're searching for a backdoor of size *k* in instance **I**

v_1	v_2	v_3	v_4	v_5	v_6
1	0	0	1	0	2
0	1	0	0	2	0
2	0	2	0	0	1
0	2	1	0	0	1

$v_4 = 0$
$v_5 = 0$
$v_6 = 1$

k = 3, c = 2

Languages and Backdoors

For every finite language Γ, strong backdoor detection to CSP[Γ] is FPT parameterized by backdoor size.

– **Recall**: variable set **X** is a strong backdoor if each assignment of X results in an instance of CSP[Γ]

– **Observation**: assume Γ has maximum arity of *c* and we're searching for a backdoor of size *k* in instance **I**

v_1	v_2	v_3
2	0	2
0	2	1

$v_4 = 0$
$v_5 = 0$
$v_6 = 1$

$k = 3, c = 2$

Languages and Backdoors

For every finite language Γ, strong backdoor detection to CSP[Γ] is FPT parameterized by backdoor size.

- **Recall**: variable set **X** is a strong backdoor if each assignment of X results in an instance of CSP[Γ]
- **Observation**: assume Γ has maximum arity of *c* and we're searching for a backdoor of size *k* in instance **I**

v_1	v_2	v_3	v_4	v_5	v_6
1	0	0	1	0	2
0	1	0	0	2	0
2	0	2	0	0	1
0	2	1	0	0	1

Constraint of arity > k+c

Arity too big;
no backdoor of size k
to CSP[Γ] can exist

k = 3, c = 2

Languages and Backdoors

For every finite language Γ, strong backdoor detection to CSP[Γ] is FPT parameterized by backdoor size.

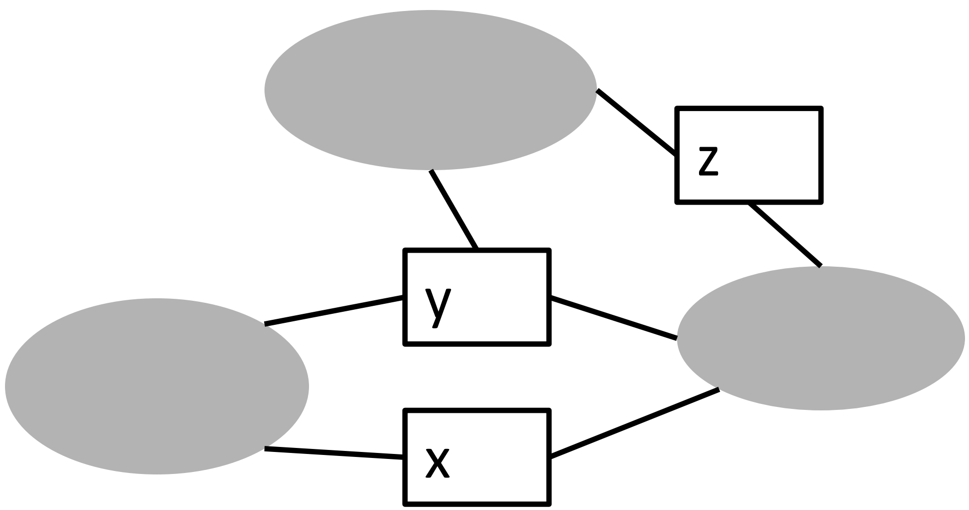
1. Check that each constraint has arity at most c+k
   - k = backdoor size, c = maximum arity in Γ
2. Proceed similarly as for Heterogeneous Backdoors for SAT
   - Start with **X** = ∅
   - Try all assignments of **X**, if we're always in CSP[Γ] then
   - If not, then branch over which of the at most k+c variables from a bad constraint goes to **X**
   - Restart

– Total runtime: $k^{O(k)} \cdot n^{O(1)}$

– Once we have such a backdoor, solving CSP is easily FPT.

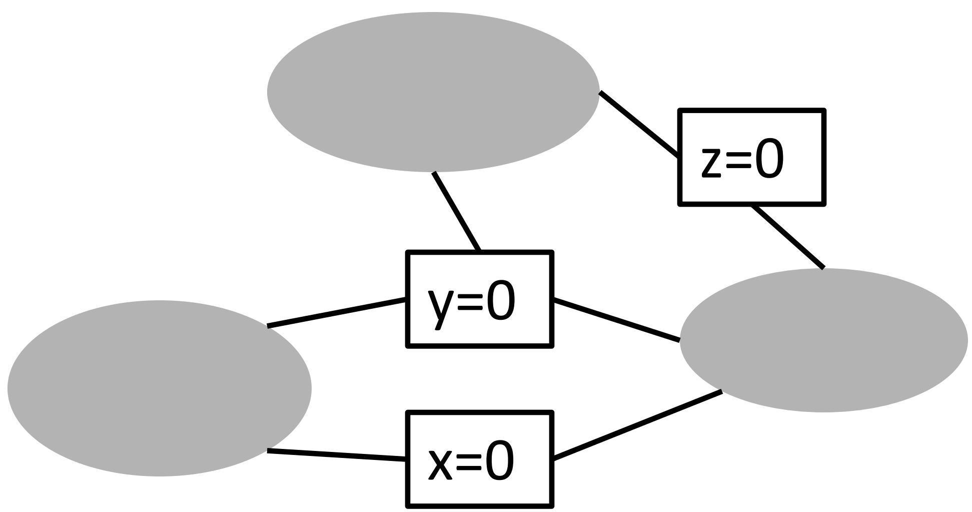
Advanced Backdoors

- Backdoors can do much more…
  - Example (Boolean CSP):



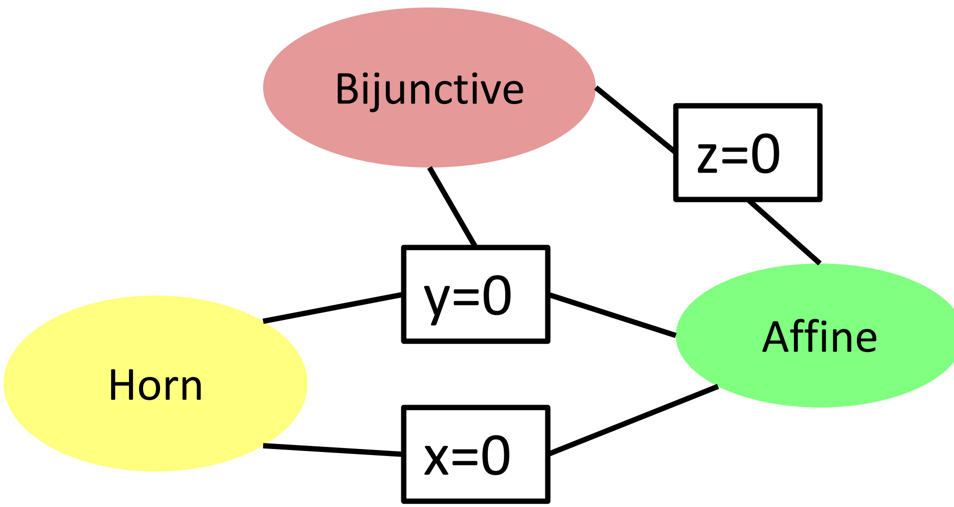
Advanced Backdoors

- Backdoors can do much more…
  - Example (Boolean CSP):



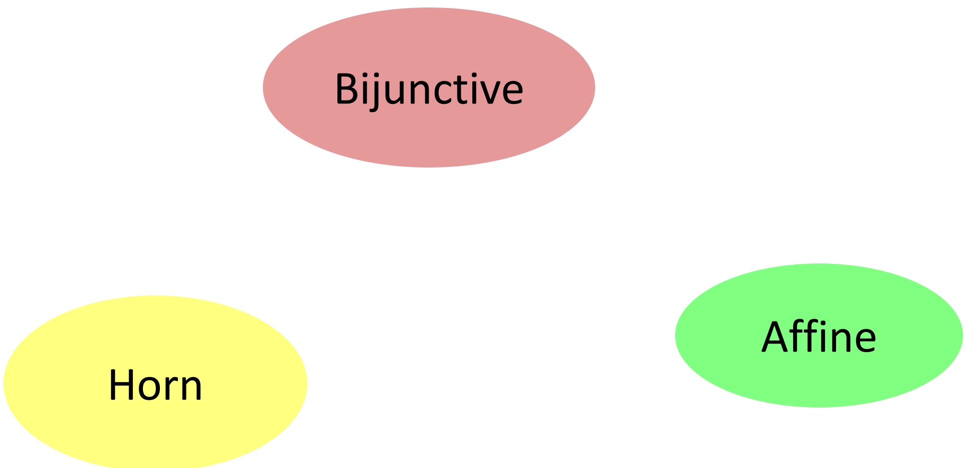
Advanced Backdoors

- Backdoors can do much more…
  - Example (Boolean CSP):



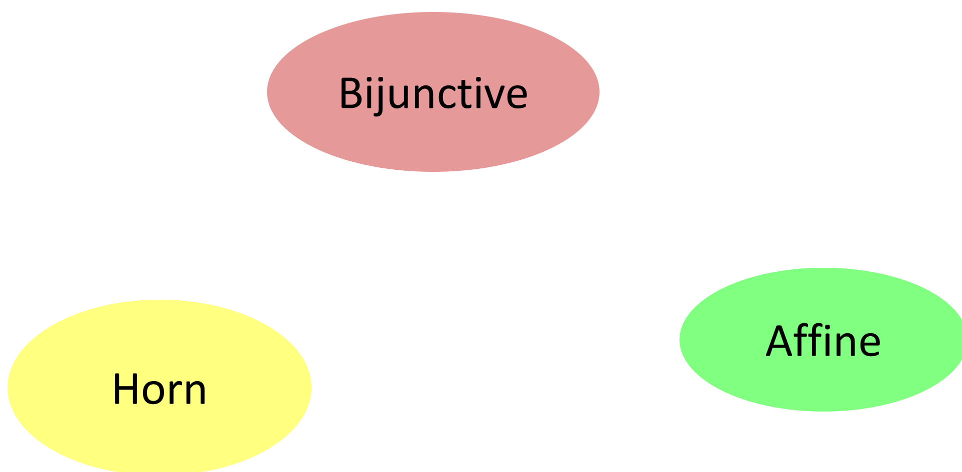
Advanced Backdoors

- Backdoors can do much more…
  - Example (Boolean CSP):



Advanced Backdoors

- Backdoors can do much more…
  - Example (Boolean CSP):

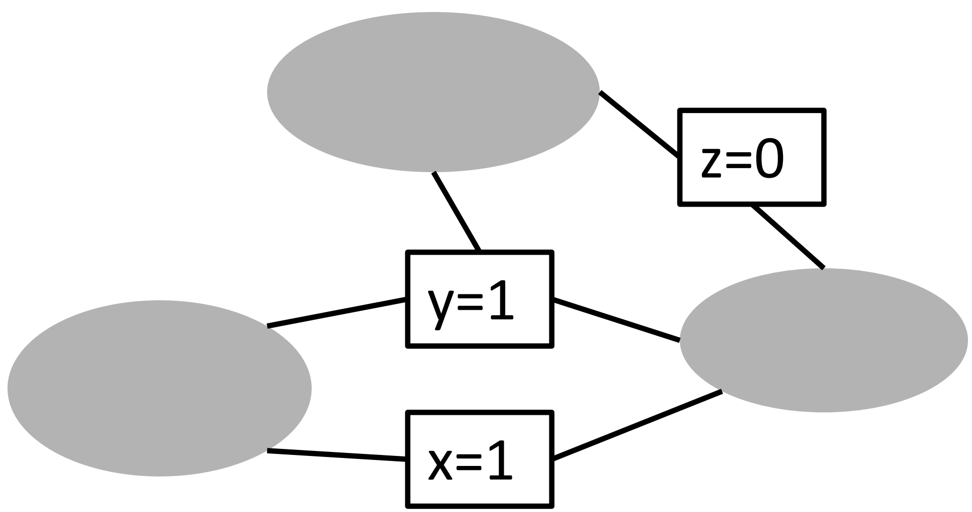


- Each connected component could belong to a different island

If we had such a backdoor, we could solve CSP in FPT time

Advanced Backdoors

- Backdoors can do much more…
  – Example (Boolean CSP):

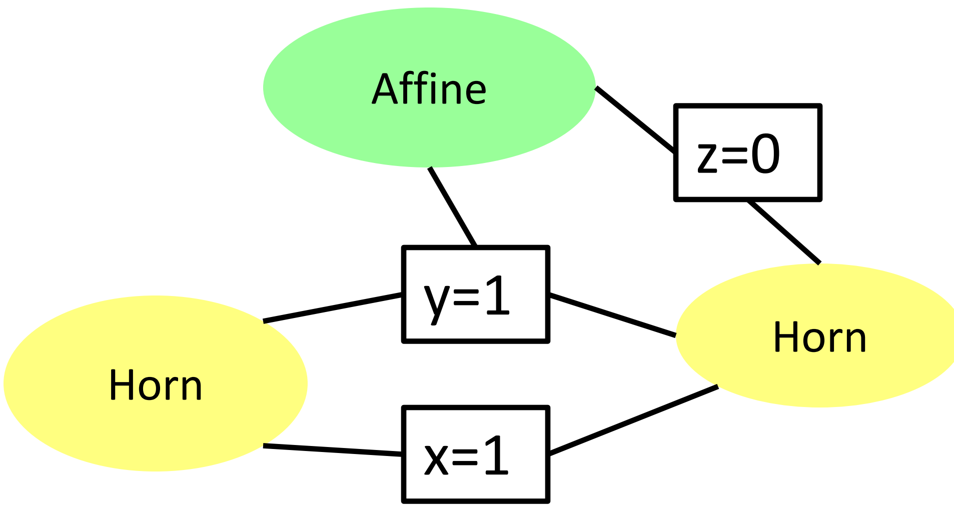


- Each connected component could belong to a different island

If we had such a backdoor, we could solve CSP in FPT time

Advanced Backdoors

- Backdoors can do much more…
  - Example (Boolean CSP):

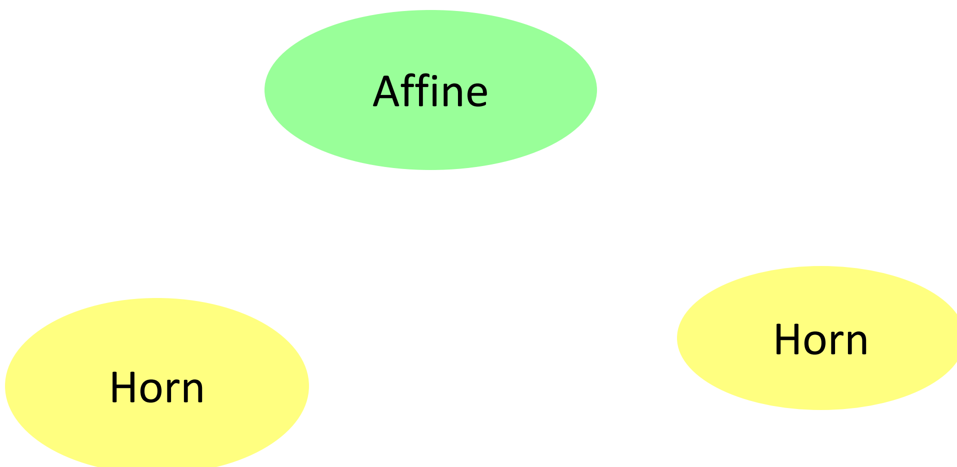


- Each connected component could belong to a different island

If we had such a backdoor, we could solve CSP in FPT time

Advanced Backdoors

- Backdoors can do much more…
  - Example (Boolean CSP):

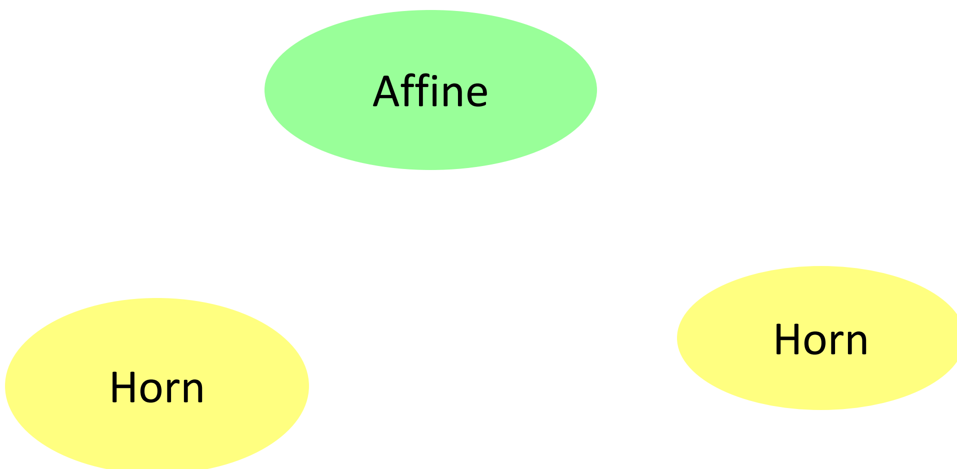


- Each connected component could belong to a different island

If we had such a backdoor, we could solve CSP in FPT time

Advanced Backdoors

- Backdoors can do much more…
  - Example (Boolean CSP):

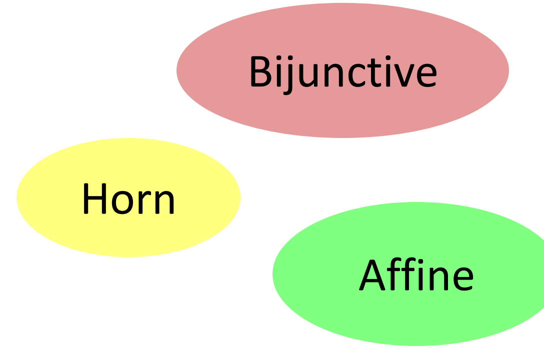


- Each connected component could belong to a different island
- Islands can change (like with heterogeneous backdoors)

If we had such a backdoor, we could solve CSP in FPT time

Formalizing

Definition: The **scattered class** $\mathbf{CSP(\Gamma_1) \oplus CSP(\Gamma_2) \oplus \ldots \oplus CSP(\Gamma_j)}$ contains all instances where each component belongs to at least one of $\mathbf{CSP(\Gamma_1), CSP(\Gamma_2), \ldots, CSP(\Gamma_j)}$.



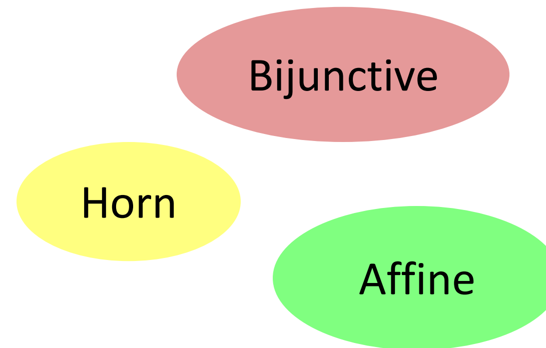
The good: backdoors to **scattered classes** are as easy to evaluate as standard backdoors

- try all instantiations
- for each, we can process every component separately

Formalizing

Definition: The **scattered class** $\mathbf{CSP(\Gamma_1) \oplus CSP(\Gamma_2) \oplus \ldots \oplus CSP(\Gamma_j)}$ contains all instances where each component belongs to at least one of $\mathbf{CSP(\Gamma_1), CSP(\Gamma_2), \ldots, CSP(\Gamma_j)}$.

The good: backdoors to **scattered classes** are as easy to evaluate as standard backdoors



Formalizing

Definition: The **scattered class** $\mathbf{CSP}(\Gamma_1)\oplus\mathbf{CSP}(\Gamma_2)\oplus\ldots\oplus\mathbf{CSP}(\Gamma_j)$ contains all instances where each component belongs to at least one of $\mathbf{CSP}(\Gamma_1),\mathbf{CSP}(\Gamma_2),\ldots,\mathbf{CSP}(\Gamma_j)$.

Bijunctive

Horn

Affine

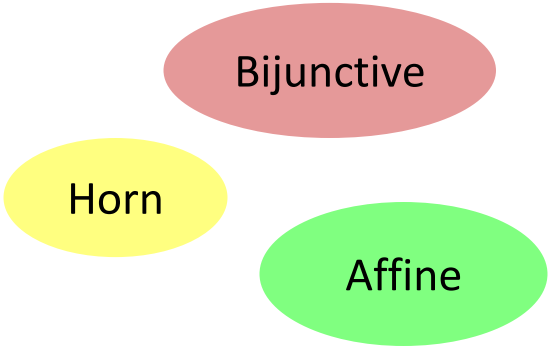
The good: backdoors to **scattered classes** are as easy to evaluate as standard backdoors

The bad: backdoors to **scattered classes** are much more challenging to find than standard backdoors

- Previously: each variable is used to kill some "bad constraints"
- Now: variables may also be used to disconnect instance; "bad constraints" no longer defined

Formalizing

Definition: The scattered class $\mathbf{CSP(\Gamma_1)\oplus CSP(\Gamma_2)\oplus \ldots \oplus CSP(\Gamma_j)}$ contains all instances where each component belongs to at least one of $\mathbf{CSP(\Gamma_1), CSP(\Gamma_2), \ldots, CSP(\Gamma_j)}$.

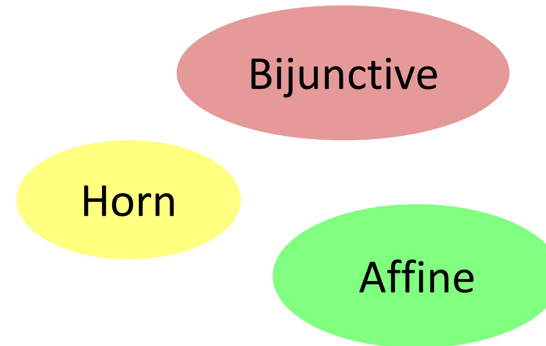


The good: backdoors to scattered classes are as easy to evaluate as standard backdoors

The bad: backdoors to scattered classes are much more challenging to find than standard backdoors

Formalizing

Definition: The scattered class $\mathbf{CSP}(\Gamma_1)\oplus\mathbf{CSP}(\Gamma_2)\oplus\ldots\oplus\mathbf{CSP}(\Gamma_j)$ contains all instances where each component belongs to at least one of $\mathbf{CSP}(\Gamma_1),\mathbf{CSP}(\Gamma_2),\ldots,\mathbf{CSP}(\Gamma_j)$.



- **The good**: backdoors to scattered classes are as easy to evaluate as standard backdoors
- **The bad**: backdoors to scattered classes are much more challenging to find than standard backdoors
- **The pretty**: backdoors to scattered classes can be arbitrarily smaller than standard backdoors

Backdoors to Scattered Classes

CSP is FPT parameterized by the size of a minimum backdoor into $\mathrm{CSP}(\Gamma_1) \oplus \mathrm{CSP}(\Gamma_2) \oplus \ldots \oplus \mathrm{CSP}(\Gamma_j)$ for any finite, tractable and conservative $\Gamma_1, \Gamma_2, \ldots, \Gamma_j$.

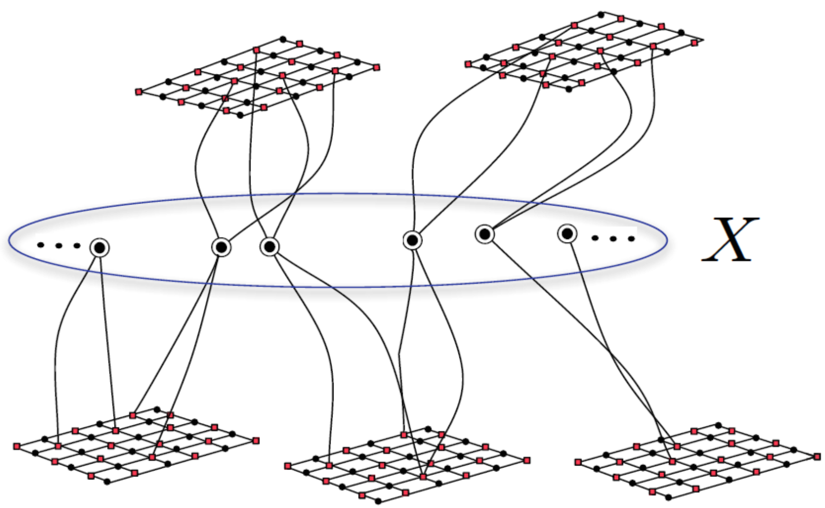
– Ganian, Ramanujan, Szeider 2016
– Classification result

Can we get *efficient* algorithms for specific languages

Large Backdoors

- Assume we have a backdoor **X** to a tractable **CSP(Γ)** which:
  - is large, but
  - has "simple" interactions with the rest of **I**

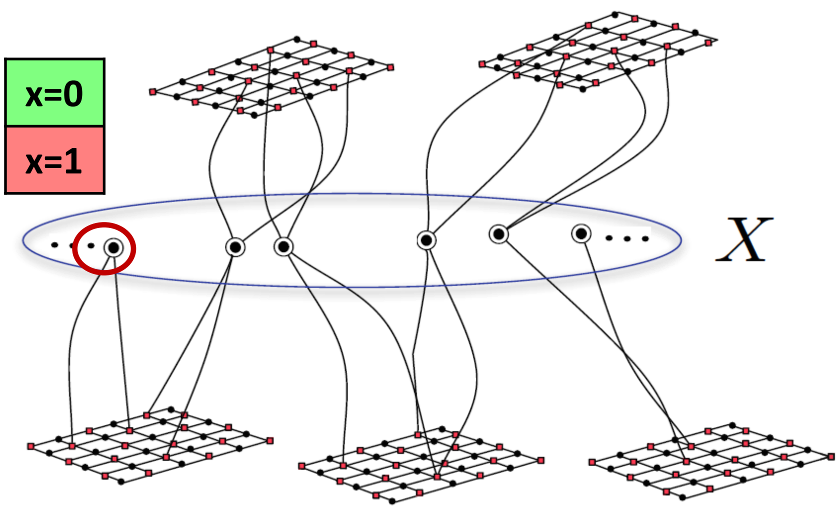
- Can we use **X** to solve **I** efficiently?
  - **cannot** try all instantiations
  - **cannot** use incidence treewidth
  - **can** use dynamic programming
    - Process backdoor variables in sequence
    - Only keep track of feasible instantiations for current pair
    - see if any satisfying instantiation survives till the end



Large Backdoors

- Assume we have a backdoor **X** to a tractable **CSP(Γ)** which:
  - is large, but
  - has "simple" interactions with the rest of **I**

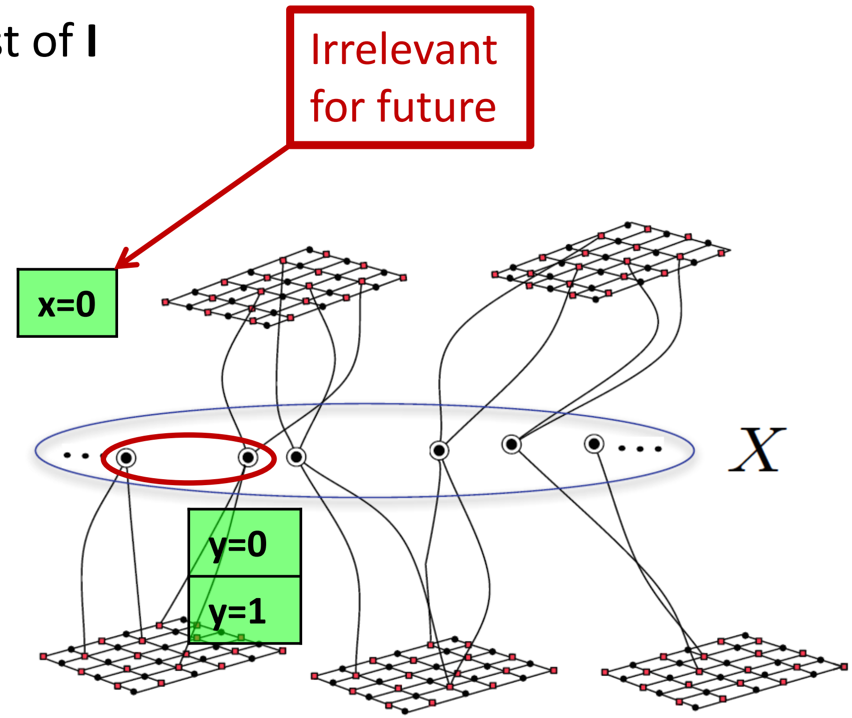
- Can we use **X** to solve **I** efficiently?
  - **cannot** try all instantiations
  - **cannot** use incidence treewidth
  - **can** use dynamic programming
    - Process backdoor variables in sequence
    - Only keep track of feasible instantiations for current pair
    - see if any satisfying instantiation survives till the end



Large Backdoors

- Assume we have a backdoor **X** to a tractable **CSP(Γ)** which:
  - is large, but
  - has “simple” interactions with the rest of **I**

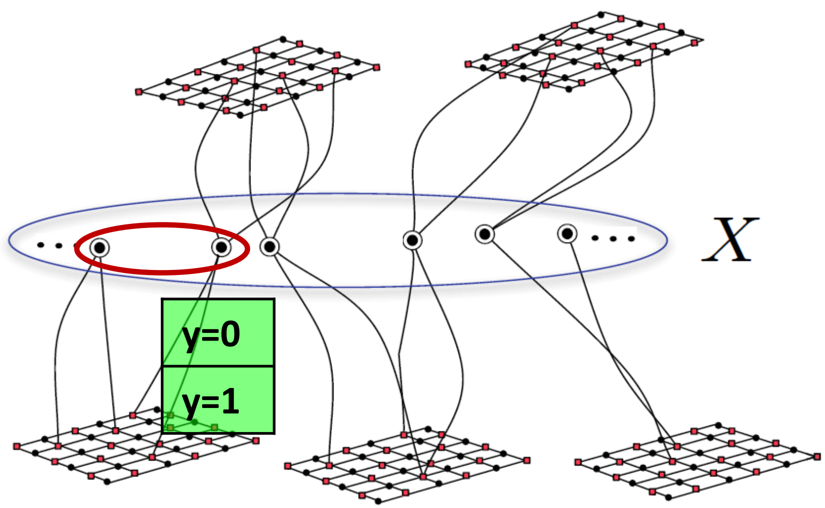
- Can we use **X** to solve **I** efficiently?
  - **cannot** try all instantiations
  - **cannot** use incidence treewidth
  - **can** use dynamic programming
    - Process backdoor variables in sequence
    - Only keep track of feasible instantiations for current pair
    - see if any satisfying instantiation survives till the end



Large Backdoors

- Assume we have a backdoor **X** to a tractable **CSP(Γ)** which:
  - is large, but
  - has "simple" interactions with the rest of **I**

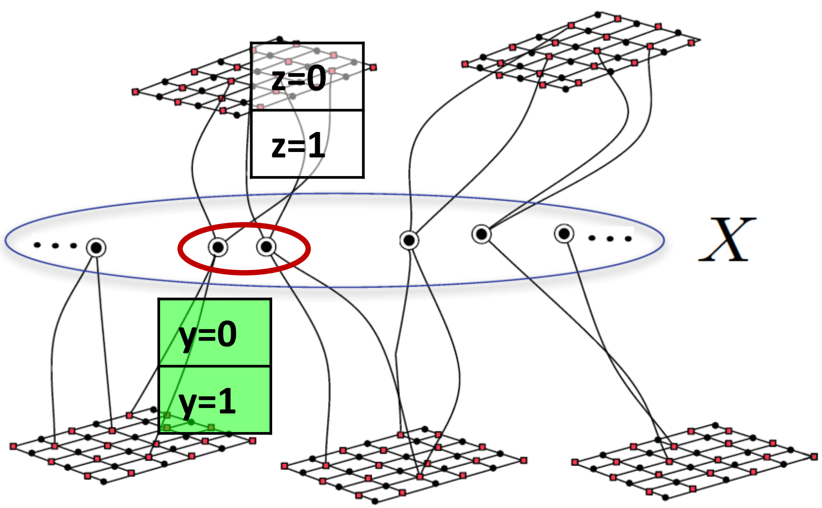
- Can we use **X** to solve **I** efficiently?
  - **cannot** try all instantiations
  - **cannot** use incidence treewidth
  - **can** use dynamic programming
    - Process backdoor variables in sequence
    - Only keep track of feasible instantiations for current pair
    - see if any satisfying instantiation survives till the end



Large Backdoors

- Assume we have a backdoor **X** to a tractable **CSP(Γ)** which:
  – is large, but
  – has "simple" interactions with the rest of **I**

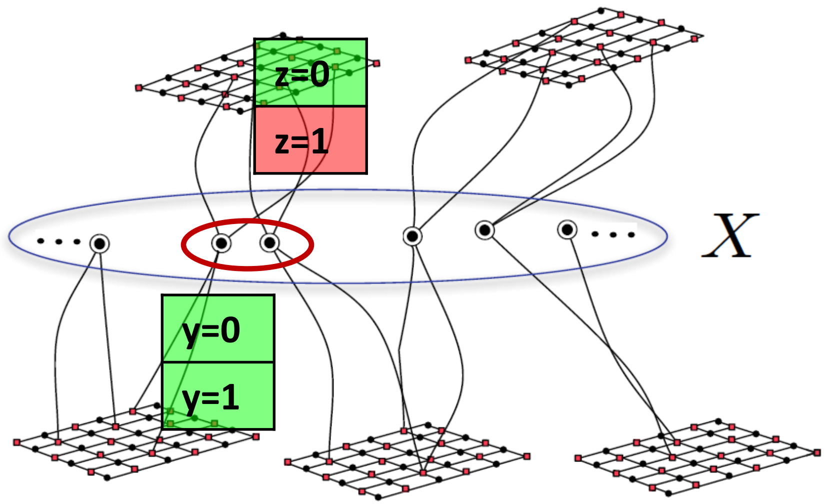
- Can we use **X** to solve **I** efficiently?
  – **cannot** try all instantiations
  – **cannot** use incidence treewidth
  – **can** use dynamic programming
    - Process backdoor variables in sequence
    - Only keep track of feasible instantiations for current pair
    - see if any satisfying instantiation survives till the end



Large Backdoors

- Assume we have a backdoor **X** to a tractable **CSP(Γ)** which:
  - is large, but
  - has “simple” interactions with the rest of **I**

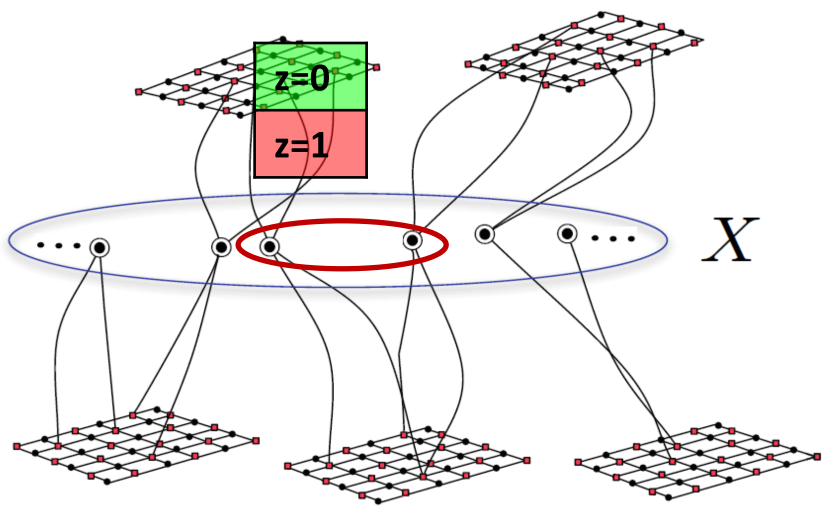
- Can we use **X** to solve **I** efficiently?
  - **cannot** try all instantiations
  - **cannot** use incidence treewidth
  - **can** use dynamic programming
    - Process backdoor variables in sequence
    - Only keep track of feasible instantiations for current pair
    - see if any satisfying instantiation survives till the end



Large Backdoors

- Assume we have a backdoor **X** to a tractable **CSP(Γ)** which:
  - is large, but
  - has “simple” interactions with the rest of **I**

- Can we use **X** to solve **I** efficiently?
  - **cannot** try all instantiations
  - **cannot** use incidence treewidth
  - **can** use dynamic programming
    - Process backdoor variables in sequence
    - Only keep track of feasible instantiations for current pair
    - see if any satisfying instantiation survives till the end

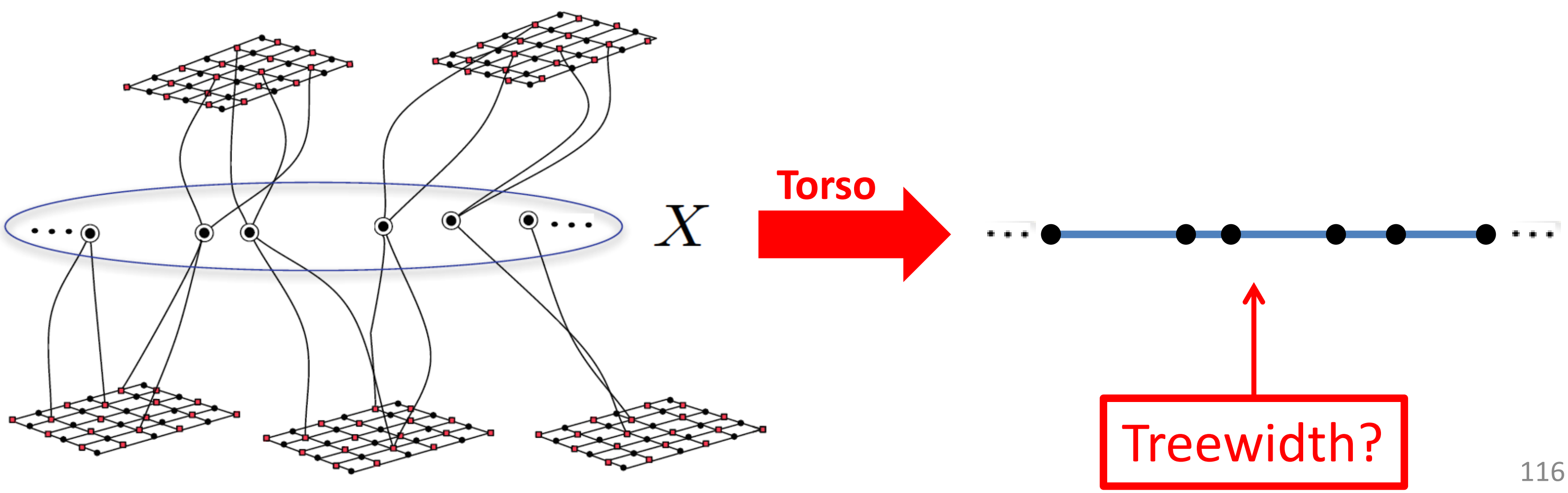


Formalizing the idea

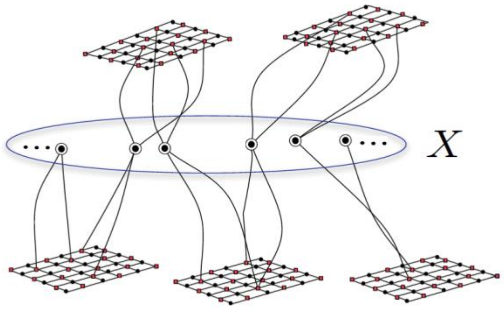
Definition: The **backdoor treewidth** w.r.t. **Γ** is the minimum treewidth of the **torso of a backdoor** to **CSP(Γ)**.

Torso of a backdoor:

- collapses everything into the backdoor
- fully captures interactions between backdoor variables



Backdoor Treewidth



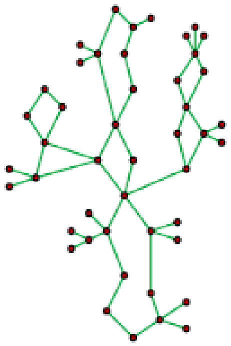
- **Evaluation**:
  A backdoor of treewidth **k** into tractable **Γ** can be used to solve **CSP** in FPT time
  - Dynamic programming (**example**)
  - Requires bounded domain (**like backdoors and treewidth**)

- **Finding:**
  Much more challenging than finding backdoors of size **k**
  - Backdoors of small treewidth need not be minimum backdoors into **Γ**
  - Instances could have large treewidth and only large backdoors
  - Even membership in XP is not obvious

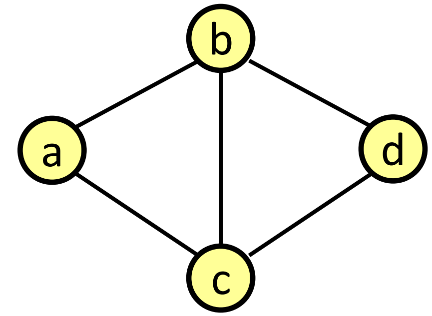
Backdoor Treewidth

Finding a backdoor to CSP(Γ) of width at most *k* is FPT for every finite language Γ.

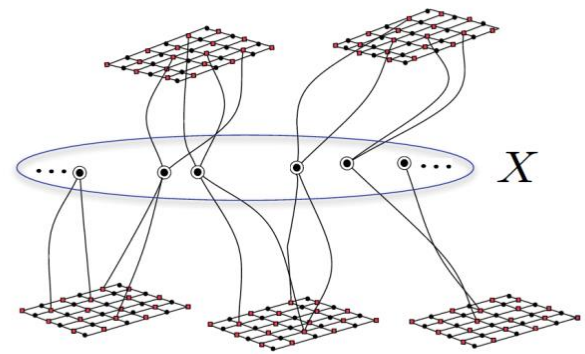
- Ganian, Ramanujan, Szeider (2017)
- Also works for SAT (e.g., backdoors to Horn) without arity restrictions

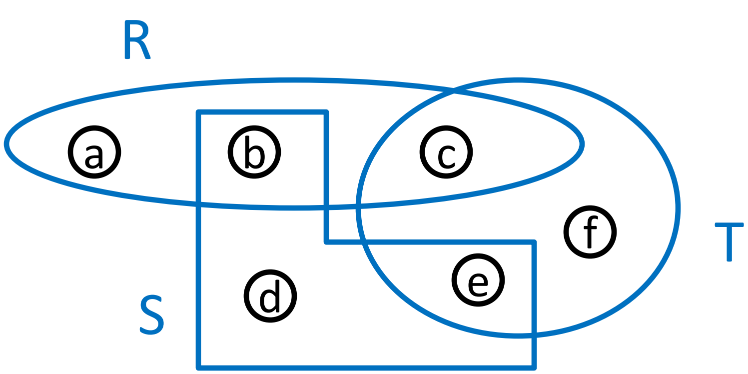


Thank you for your attention



Questions?

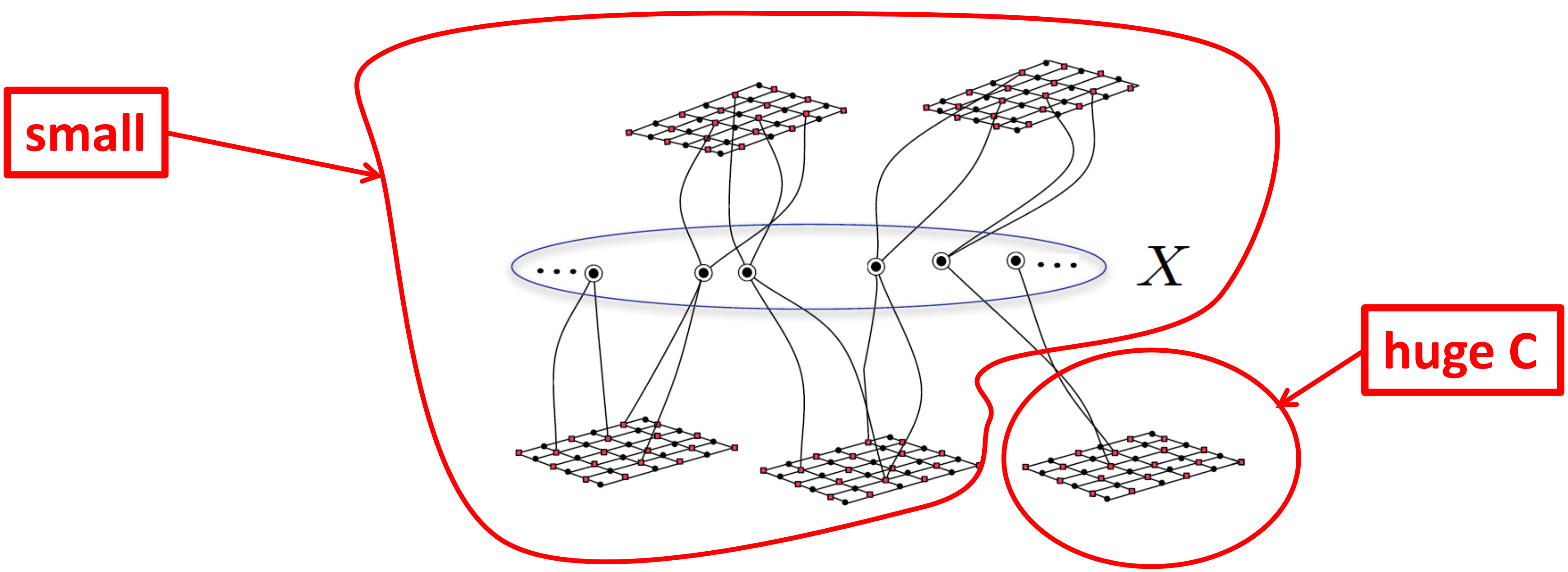




Finding small-treewidth backdoors

- First task: dealing with **nice** instances
  - an instance **I** is **nice** if at least one of these hold:
    - **I** has small incidence treewidth, or
    - **I** has a small-treewidth backdoor **X** with precisely one connected component **C** such that **I-C** is small

$\leq f(k)$



Why “nice”?

Nice instances are easy to solve

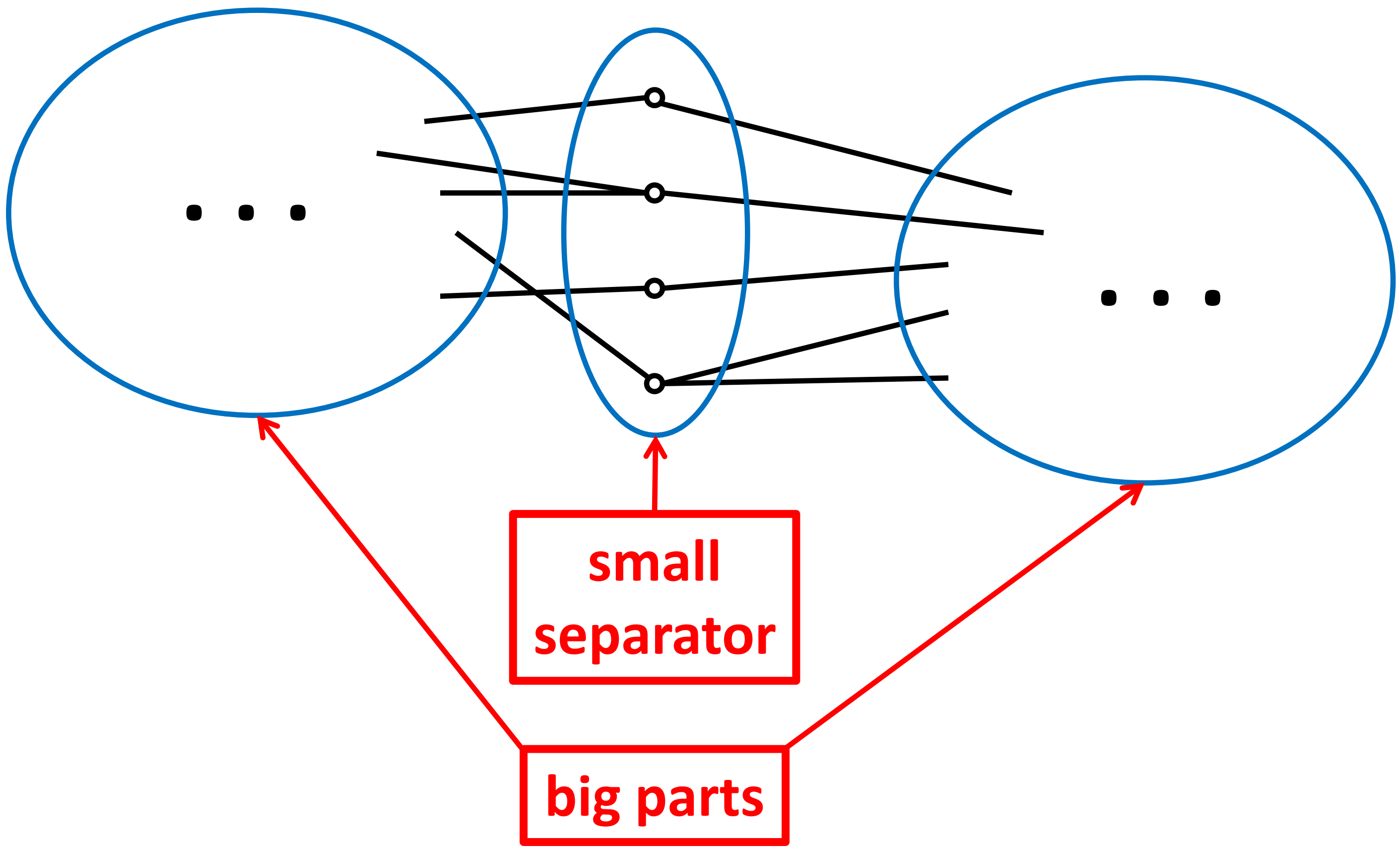
- If incidence treewidth is small…
  - we can use, e.g., **Courcelle’s Theorem** to find a small-treewidth backdoor
  - (we could also solve the instance directly if we wanted to)

- If everything outside of **C** is small…
  - then everything outside of **C** is actually a small backdoor

Nice instances will also be important later on

Dealing with ugly instances

- ugly instances have a **good separation**
  (assuming they have a small-treewidth backdoor **X**)

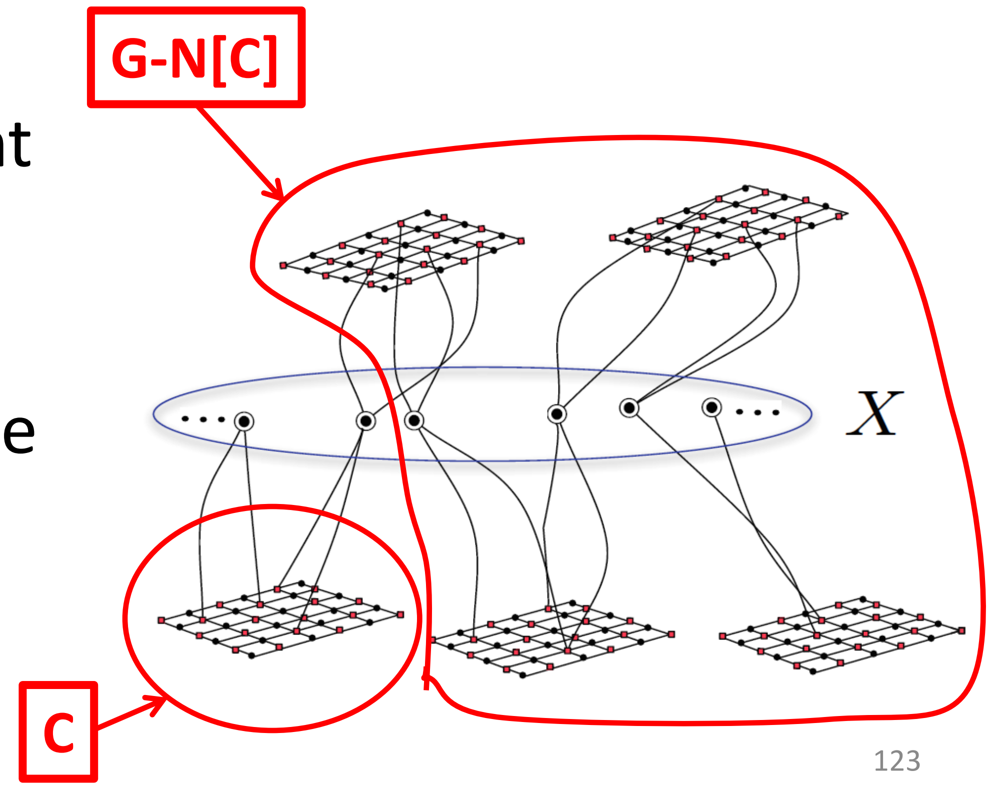


Dealing with ugly instances

- ugly instances have a **good separation** (assuming they have a small-treewidth backdoor **X**)

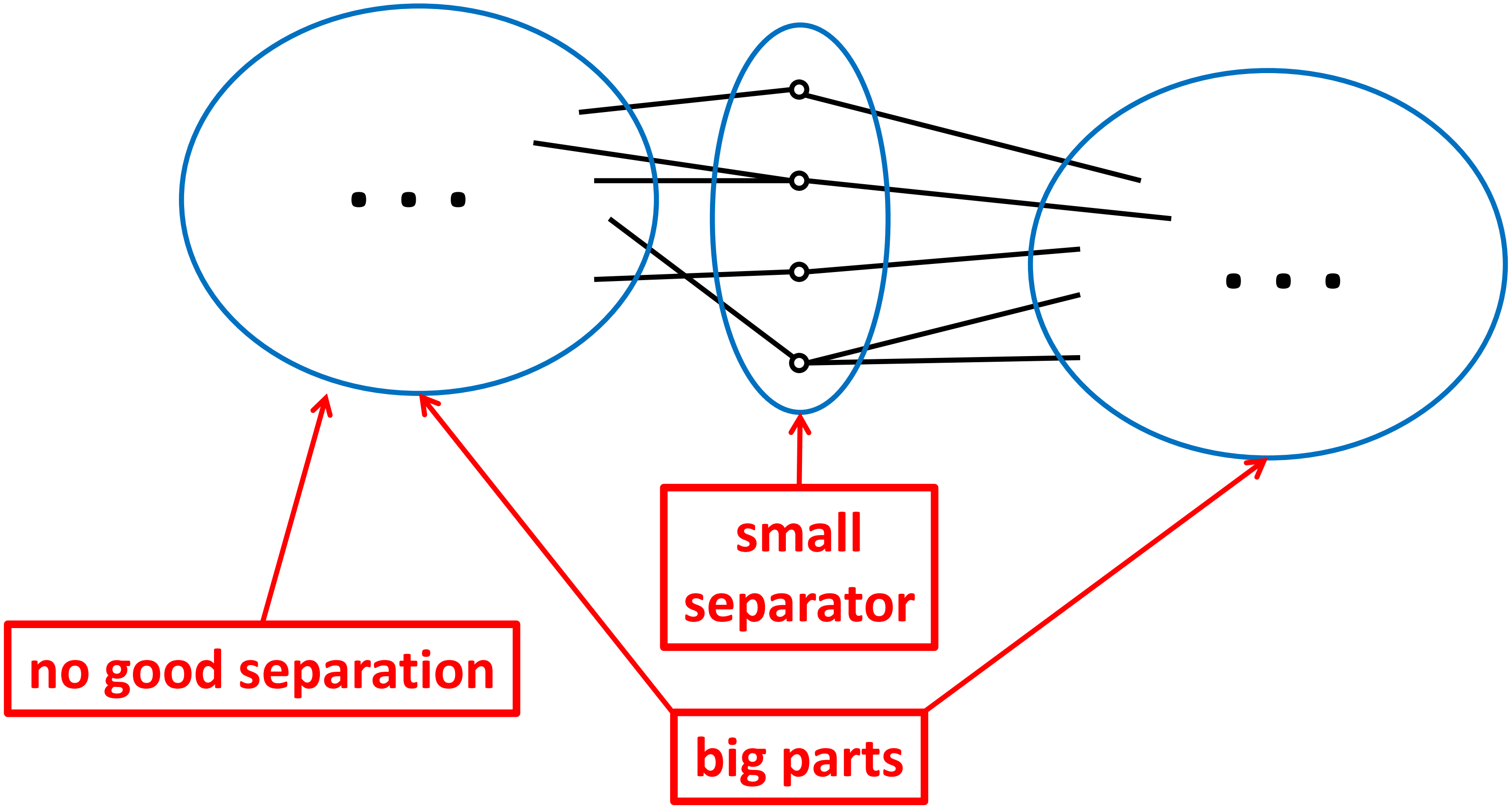
Why?

- Find biggest component **C** in **G-X**
- If **C** or **G-N[C]** is small then the instance is nice
- Otherwise we have a **good separation**



Finding good separations

- Using standard techniques, we find a "***left-most***" **good separation** in FPT time

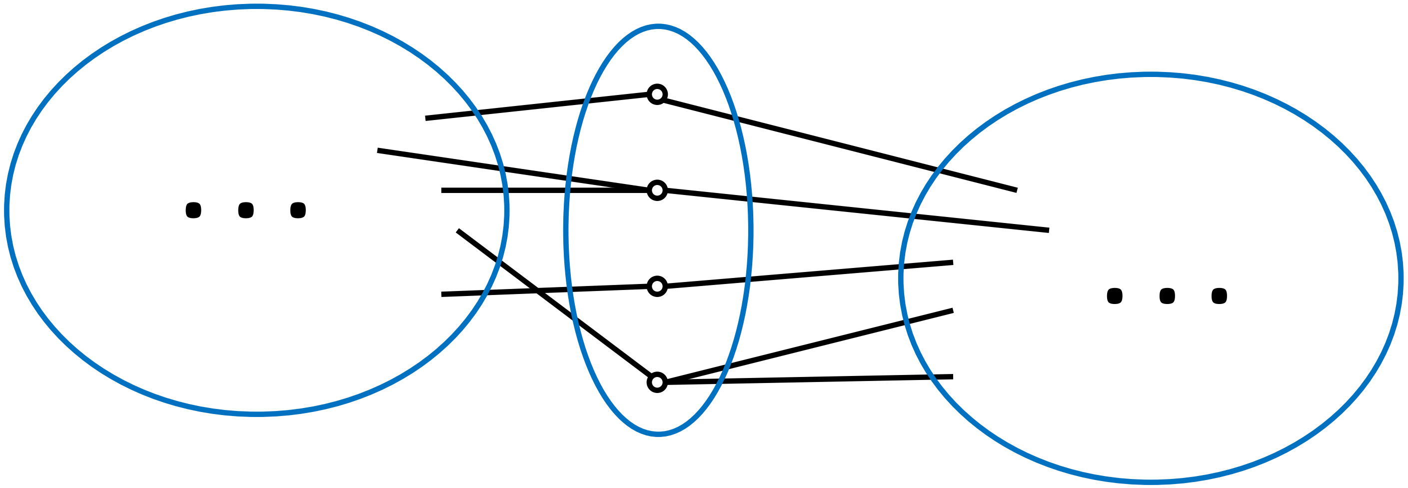


Finite State machinery

- Our next goal will be to replace the left side with a **small representative**
  - Requires development of finite state machinery for CSPs capturing contribution to a small-treewidth backdoor
  - End result: small set **Q** of **small representatives** for all possible parts on one side of a separator

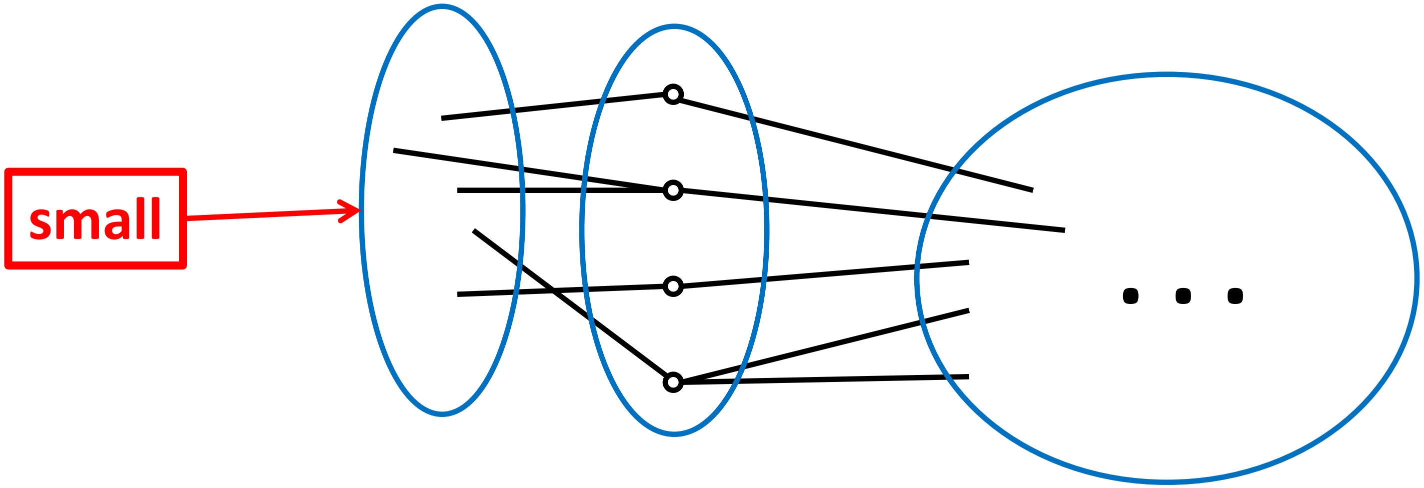
Finite State machinery

- Our next goal will be to replace the left side with a **small representative**



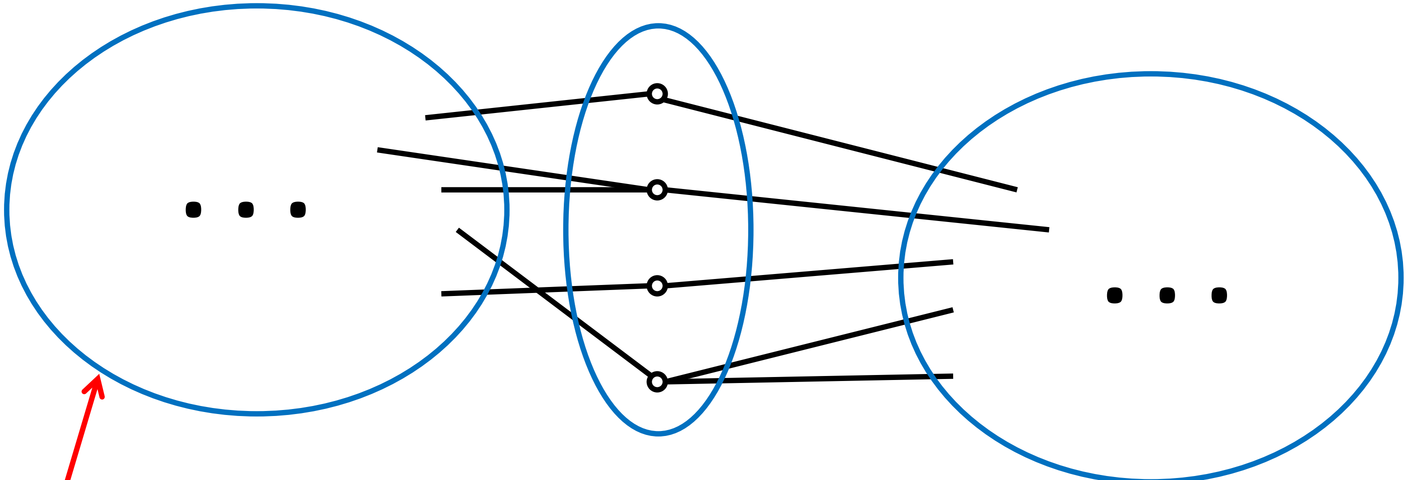
Finite State machinery

- Our next goal will be to replace the left side with a **small representative**



- New instance strictly smaller but equivalent
  - We now restart with new smaller instance

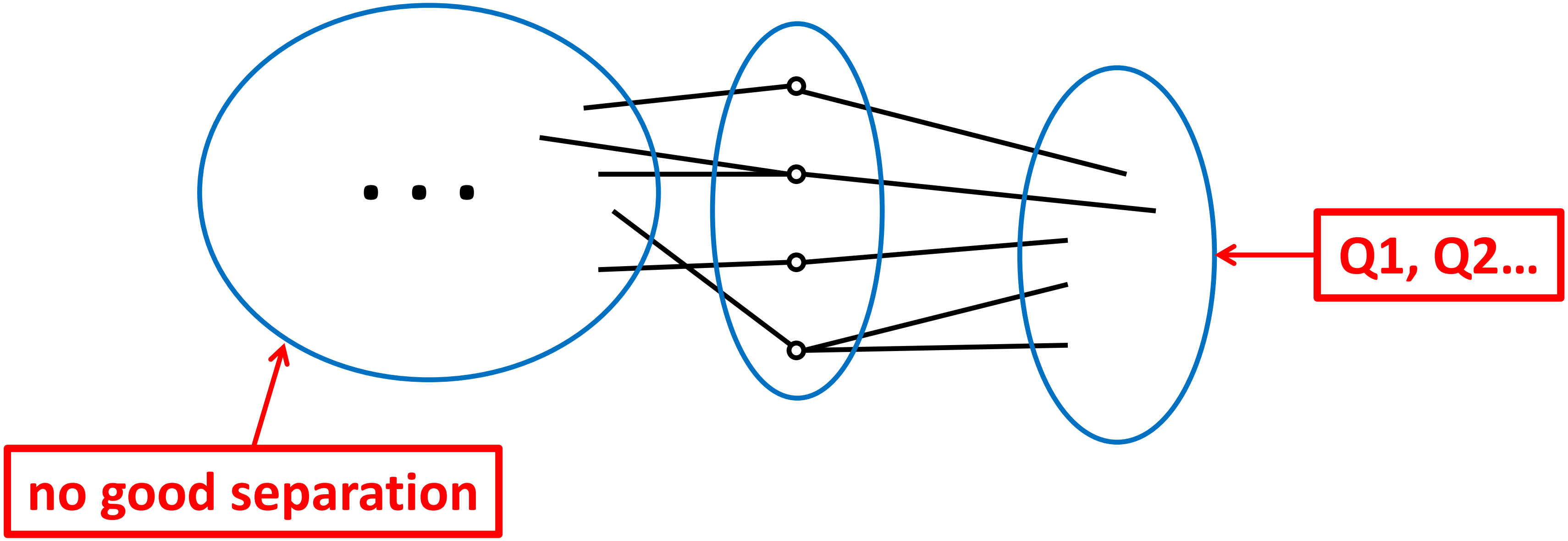
Choosing the right representative



no good separation

- How to choose the correct **representative** from **Q?**
  - Test the left side against all possible **representatives**

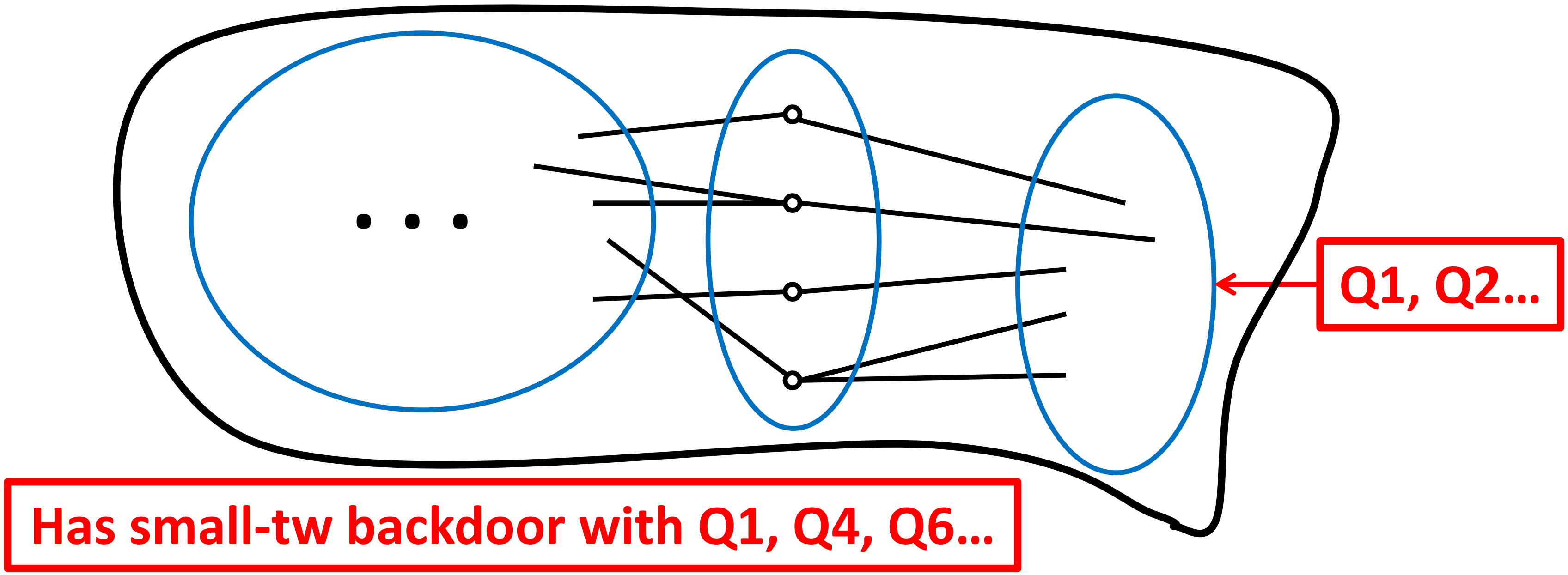
Choosing the right representative



- How to choose the correct **representative** from **Q?**
  - Test the left side against all possible **representatives**
  - Can prove that resulting instances contain no **good separation** (w.r.t. slightly bigger constants)

➡ they are nice ➡ can determine how left side interacts with all possible representatives

Choosing the right representative

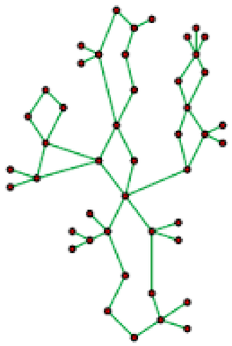


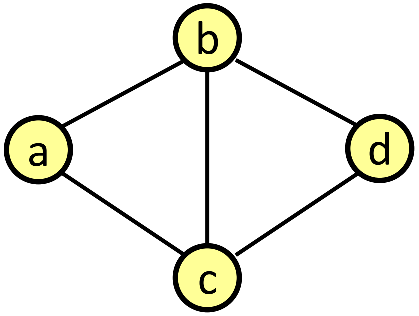
- How to choose the correct **representative** from **Q?**
  - Pick **representative** for left side which interacts the same way with all **representatives** in **Q**

Final Recap

Finding a backdoor to CSP(Γ) of width at most *k* is FPT for every finite language Γ.

- If **I** is **nice**, directly find a small-treewidth backdoor
- Otherwise, try to find a **left-most** **good separation**
  - if it doesn't exist then there's no small-treewidth backdoor
- Determine which **representative** fits for the left side
- Use it to obtain an equivalent but smaller instance
  - Restart on new instance





Thank you for your attention

Questions?